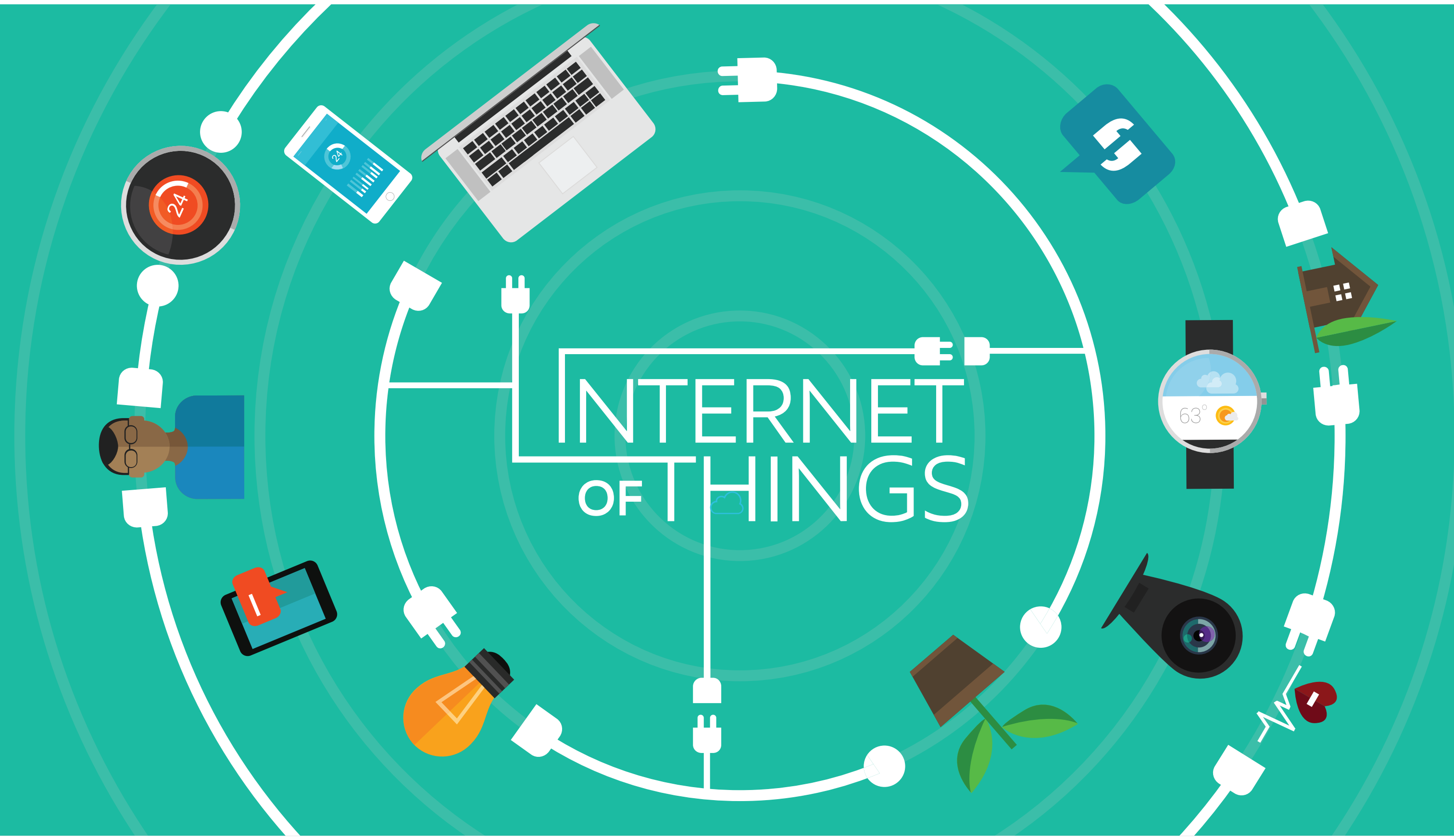
Editors: Barry I. Schneider, bis@nist.gov | Gabriel A. Wainer, gwainer@sce.carleton.ca



Discrete-Event Modeling and Simulation for Embedded Systems

Daniella Niyonkuru and Gabriel A. Wainer | Carleton University

In the past few decades, computers have ascended to a position of prevalence in our lives. In fact, the machines in our homes and offices now outnumber the people who use and live with them. Most of these computers go unnoticed, however—these embedded computing systems are composed of tightly coupled hardware and software designed for a specific purpose. Embedded systems are found in a wide range of applications, from common electronic devices (cell phones, portable video games, camcorders) to home appliances (microwave ovens, home security systems, lighting systems) to business equipment (cash registers, alarm systems, card readers) to automobiles (cruise control, antilock brakes, fuel injection).

Embedded systems have specific characteristics that differentiate them from other computing systems[1]:

- They usually execute one program repeatedly, and they perform one particular task during their lifetime.
- Many of them must continually react to changes in the system's environment and compute certain results in real time. Responses after a strict deadline could result in catastrophic consequences—for example, an airplane's flight controller must respond correctly within the required deadline, or the result could be serious injuries or death.
- They often cost a few dollars, fit on a single chip, process data in real time, and consume minimum power.

Embedded computing systems have grown not only in popularity but also in complexity. But are current software development methods still suitable? Can they cope with this increased complexity? In a word, no—current development methodologies don't measure up to the task. Their shortcomings make software the most costly and least reliable part of an embedded system, with deficiencies originating from inconsistencies in the development cycle and problems with system verification because no single robust development framework offers an optimal solution. Formal methods have shown promising results in dealing with these issues, facilitating system verification through their strict mathematical foundations. However, most formal methods don't scale up well, and they usually don't consider the physical environment that the embedded system controls, resulting in expensive testing efforts with no guarantees of bug-free products. Instead, systems engineers usually rely on modeling and simulation (M&S) techniques that provide the means to solve these problems, even though most M&S methods are at best semiformal and don't have the robust foundation that's essential to proving the properties inherent to a system.

Here, we describe a new methodology based on discrete-event modeling and simulation that lets us model the physical system in which the embedded system works and the software that controls the application. Our approach offers a systems-theoretical methodology that uses an engineering-based approach to integrate multiple views at different levels of abstraction.

Embedded Software

The embedded systems industry generally uses design approaches based on earlier experience with similar products and, in many cases, on ad hoc techniques because the emphasis is on backward compatibility to reduce costs. However, this design rigidity leads to overly complicated systems and system management, along with skyrocketing cost. In the automotive industry, for example, cars now have an increasing number of electronic control units that have escalated software complexity to the extent that current development tools make it difficult to build reliable systems.

New abstract and visual design methods attempt to deal with this complexity and increase embedded systems' reliability and performance. Formal methods in particular provide mathematical models and use transformations, formal proofs, and validation of model specifications to prove system properties. Timed Automata (TA), for instance, provides a sound theory to specify models using a timed state-based notation. System verification is easy in this case because states' space exploration or tools that rely on computation tree logic such as UPPAAL can ensure that system properties are correct. Research in this area has advanced steadily, but most formal methods work best on medium-scale designs with well-defined interfaces. Another restriction is that formal models of real-time controllers (which usually have discrete variables) must interact with models of the physical environment (which are better modeled with continuous variable methods such as differential equations). Building such hybrid models is anything but simple.

Consequently, systems engineers often rely on M&S because it provides a flexible and risk-free method for analysis, regardless of application size (including external environment). Products built using M&S are of better quality and have an overall reduced cost because the risk-free virtual environment allows extensive verification and testing. This is an especially useful approach when you consider that verification under actual operating conditions isn't cost-effective and could be impractical (or impossible) in some cases. However, its lack of a mathematical foundation means that M&S isn't as robust as formal methods: you can't guarantee design properties with it. So how do we find a solution that provides the benefits of both formal methods and M&S? Will that solution respond to the increasing complexity and heterogeneity of embedded systems?

Our research team focuses on how to bridge formal methods and M&S to analyze embedded systems and study their interaction with the physical environment, all while making the original models become part of the target platform. To achieve this goal, we introduce a methodology called discrete-event modeling of embedded Systems (DEMES), based on a mathematical M&S theory called DEVS (discrete-event system specification).[2]

DEMES

DEMES uses M&S in its initial stages, replacing models with hardware surrogates and new software components without altering the original models. This transition happens in incremental steps, incorporating models in the target environment after thorough testing in the simulated platform, which allows model reuse throughout the process.

DEMES combines the advantages of a practical approach with the rigor of a formal method, in which you consistently use the same models throughout the development cycle following the DEVS formalism. DEVS is an increasingly accepted framework that provides an abstract and intuitive way of modeling, independent of underlying simulators, hardware, and

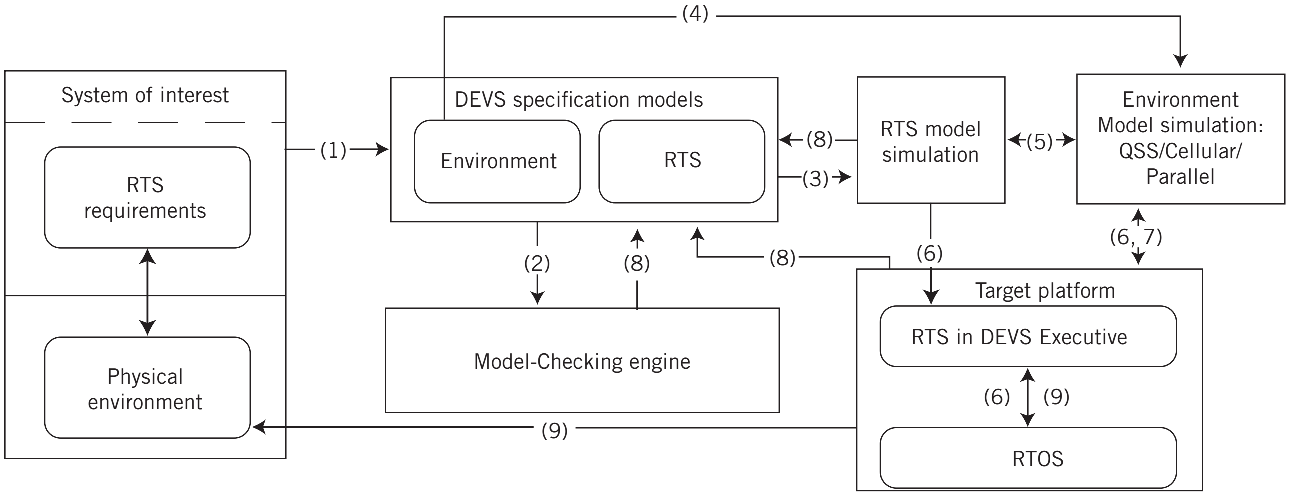


Figure 1. Discrete-Event Modeling of Embedded Systems (using DEVS). A designer starts (1) by modeling the system with formal specifications; these models (2) are then converted into a DEVS representation, transformed into timed automata, and verified using model-checking tools. In parallel with this formal verification phase, (3) the same models are used to test the components in a simulated DEVS environment. The (4) physical environment can also be simulated together with the (5) embedded system model under particular loads. Instead of obtaining general answers for all possible cases, we can simulate an individual submodel under specific conditions. Then, we deploy (6) incrementally these tested submodels into the target platform. Most of the testing phase (7) can be done with simulation, even if the hardware is unavailable. With DEMES, design changes (8) are done incrementally in a spiral cycle, providing a consistent set of apparatus throughout the development cycle.

middleware. It's based on a hierarchical and modular construction of models, which fits our needs because we define models at different levels of abstraction independently and later integrate them into a hierarchy. Indeed, the DEVS formalism decomposes complex system designs into basic (behavioral) *atomic* models and composite (structural) *coupled* models, following a precise ruleset to define state changes in modeled systems.[2]

DEVS is suitable for embedded systems because it provides a rich structural representation of components and a formal means for explicitly specifying their timing. It can also be used to model the system environment. DEVS is the most generic discrete-event formalism, and most existing real-time techniques (from finite state machines to statecharts) are transformable to it. Thus, developers can conveniently combine advanced models with different methodologies.

Figure 1 shows the DEMES architecture. A designer starts by modeling the system of interest using formal specifications, and these models are then converted into a DEVS representation, transformed into timed automata, and verified using model-checking tools. In parallel with this formal verification phase (which can take a long time, especially if state explosion happens during formal verification), the same models are used to test the components in a simulated DEVS environment. The physical environment can also be simulated with the embedded system model under particular loads. Instead of obtaining general answers for all possible cases (like those provided by model-checking), we can simulate individual submodel behaviors under specific conditions before incrementally deploying these tested submodels into the target platform. If the hardware isn't readily available, software components can still be developed incrementally and tested against a model to verify viability and make early design decisions. As the design process evolves, both software and hardware models can be refined by progressively setting checkpoints in real prototypes. The executive allows dynamic model execution and static and dynamic task scheduling. At this point, those parts that are still unverified in the formal and simulated environments are tested, increasing the engineer's confidence in the implemented system. Most of the testing phase can be done via simulation (with faster-than-real-time performance), even if the hardware is unavailable. With DEMES, design changes are done incrementally in a spiral cycle, providing a consistent set of tools throughout the development cycle. The cycle ends with the embedded system fully tested and every model deployed on the target platform.

This approach has various advantages when compared to existing methodologies, especially because most M&S methods are semiformal and don't provide direct model continuity. For instance, researchers[3] comparing DEVS and UML-RT found that features such as time, scheduling, and performance coded with UML constructions aren't formally defined, whereas formal modeling methods such as DEVS provide sound syntax/semantics for structure, behavior, time representation, and composition. Model continuity refers to preserving the model specification as much as possible through the development process, and it's usually missing in common approaches. One approach[4] showed an example of how to manage the development and testing complexity of a leader-follower robotic system by applying model continuity in a DEVS-based process. With DEMES, we place model continuity at the core by using the same models throughout the entire development process.

Model Checking in DEMES

Model-checking lets designers verify model correctness and eventually produce formally correct software. One advantage of executable models is that they can be deployed to the target platform, thus providing the opportunity to use the controller model not only for simulations but also as the actual code executing on the target hardware. Thus, the verified model is itself the final implementation executing in real time. This prevents any new errors that might appear during transformation of the verified models into an implementation, hence guaranteeing a high degree of correctness and reliability.

To verify DEVS models, we use a class of rational time-advance DEVS called RTA-DEVS,[5] transforming these RTA-DEVS models into equivalent TA that are then used to formally verify desired properties using the UPPAAL model checker tool. RTA-DEVS was introduced to provide the modeler with a formalism that's both expressive and sufficient to model complex system behavior while being verifiable through formal model-checking techniques. RTA-DEVS is a subclass of DEVS that restricts the time advance function to nonnegative rational numbers and the elapsed time in a state used in the external transition to be a nonnegative rational number. These restrictions mean that we'll have nonnegative rational constants in the resulting TA model and ensures termination of the reachability analysis algorithms implemented in UPPAAL.

In previous work,[6] we presented a case study using a controller for an e-puck robotic application and demonstrated this approach's practicality.

Simulation

The DEVS formalism proposes a framework for model construction and defines an abstract simulation mechanism that's independent of the model itself. This mechanism provides a high-level implementation detail for the DEVS framework and can be feasibly implemented by computer software.

The simulation/execution process involves two primary subsystems: the modeling subsystem and the runtime subsystem. The modeling subsystem allows the modeler to define atomic and coupled models by extending the basic model classes. The runtime subsystem is hidden to the modeler and contains execution classes associated with modeling classes that execute specific algorithms (defined by the abstract simulator) to render model behavior. Although the modeling subsystem remains the same, the execution subsystem varies to allow faster-than-real-time simulation, real-time simulation, or hardware-in-the-loop simulation. Therefore, the execution engine used for physical environment models will differ from the one associated with real-time simulation because the latter ultimately has to allow models to run on the target hardware.

We have different simulation tools dedicated to real-time simulation as well as faster-than-RT simulation. The CD++ toolkit, in particular, is a DEVS-based framework that lets users run different types of simulation.[7] It's especially useful for environment model simulation and supports QSS (quantized state systems) and cellular and parallel DEVS:

- QSS approximates partial differential equations. Various efforts have integrated continuous models and DEVS, but QSS theory showed how to approximate hybrid systems through DEVS, and QSS's main advantages (controlled error, performance, parallelism, and so on) still hold in the presence of discontinuities.[8]
- Cellular models[9] represent physical systems as lattices of elements (each including a simple computing apparatus) that can reproduce a real system's complete behavior. Cellular computing[10] has received a big push, and many researchers now use these methods.
- Parallel discrete-event simulation (PDES)[11] deals with executing discrete-event simulations on multiple interconnected processors. Various existing methods (optimistic, conservative, and so on) are starting to appear in the DEVS methodology.[12]

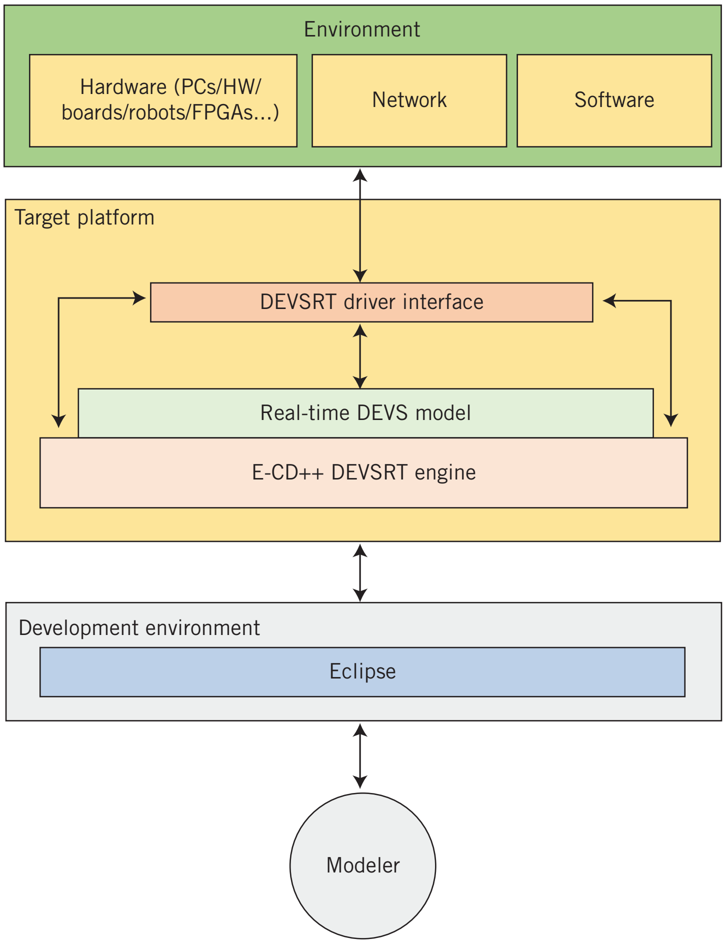


Figure 2. DEMES embedded framework. The embedded platform with the external environment in this layered approach represents cross-platform development. The modeler uses a high-level DEVS language combined with C++ to build the application.

All of these methods are useful to build physical environments models that interact with the embedded system model and to explore diverse scenarios in a risk-free environment.

Figure 2 illustrates our DEMES embedded framework. The environment supports embedded system modeling by converting the virtual time-advance functions to real time and providing a real-time execution platform to verify such models. Specifically, the figure highlights the embedded platform with the external environment in this layered approach to represent cross-platform development. The modeler uses a high-level DEVS language combined with C++ code if needed, which provides the application layer. The Eclipse IDE layer also allows for the graphical development of models.

The DEVSRT engine interprets and executes these models. To allow for direct replacement of models with external entities, I/O ports implement the formal interfacing mechanism of DEVSRT in the driver interface layer, which enables communication with the target hardware's components. The user models and the driver objects run on the target platform. If the user defines specific drivers to communicate or gather data from the external environment or external simulations, the models will also be able to process this data according to their specification.

Case Study: A Line-Following Robot Model

The system of interest for our case study application is a line-tracking robot that has a light sensor facing the ground and absorbing the light reflected off a small ground surface. The controller considers a medium percentage of reflected light as a detected path and initiates the robot to move forward. When the robot goes off track and doesn't pick up a path trail, it stops, turns, and tries to detect a trail again. The robot can also receive manual signals to start and stop.

As shown in Figure 1, the first step in the DEMES-based development cycle is to specify a model for the system of interest using DEVS. Figure 3 illustrates the resulting DEVS model hierarchy for this example.

The line-tracking robot's top model consists of three coupled models (sensor unit, control unit, and movement unit) and two input ports (MOVEL_OUT and MOVER_OUT). LIGHT_IN is the input port through which light sensor values are read, and START_IN is for start/stop commands. We use the output ports to send commands to the robot's left and right motors.

The sensor unit contains an atomic model—the light sensor—which reads the amount of light reflected and transmits those readings to the control unit. In this coupled model, the sensor controller activates or stops the light sensor (sctrl_start_out), receives the light sensor readings (sctrl_light_in), and sends messages to the movement controller (sctr_mctrl_out) specifying whether the robot is on track, off track, or reached the destination. When the robot reaches its destination—that is, the light sensor reads an all-dark surface—the sensor controller sends a "stop reading" command to the light sensor (sctrl_start_out) and a stop signal to the movement controller.

The movement controller receives on/off-track and stop signals from the sensor controller

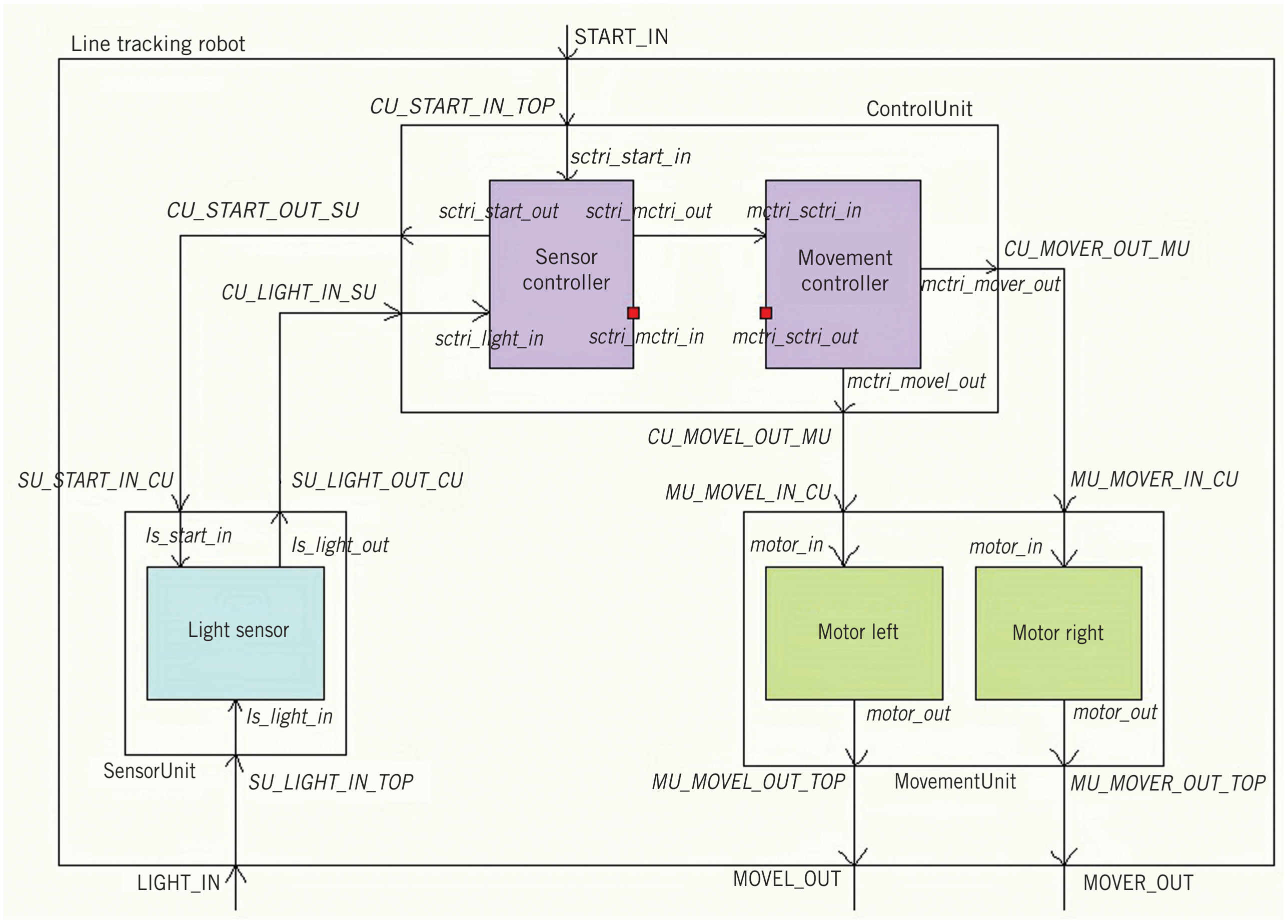


Figure 3. Model hierarchy diagram for line-following robot case study. The top model consists of three coupled models (sensor unit, control unit, and movement unit), two input ports (LIGHT_IN and START_IN), and two output ports (MOVEL_OUT and MOVER_OUT).

(mctrl_sctrl_in) and sends appropriate commands to the motors (mctrl_movel_out and mctrl_mover _out). The movement unit is made of two atomic models: motor left and motor right. Essentially, it's a collection of the robot's actuators that move in response to commands received from the control unit. The motor models control the robot treads: they can only move forward, reverse, or stop according to the signals they receive from the control unit.

The model we've just described indicates the robot's hierarchical structures, so let's now look at how to build one of those components using the DEVS formal specification. As mentioned earlier, the control Unit model has two atomic models, the sensor and movement controllers. We formally define the control unit as follows:

CM = <*X*, *Y*, *D*, {*Md*}, *EIC*, *EOC*, IC, *select*>,

where

X = {(*CU_START_IN_TOP*, *N*); (*CU_LIGHT_IN_SU*, *N*)}

Y = {(*CU_START_OUT_SU*, *N*); (*CU_MOVEL_OUT_MU*, *N*); (*CU_MOVER_OUT_MU*, *N*)}

D = {*Sensor Controller*, *Movement Controller*}

Md = {*M*(*sensor controller*), *M*(*movement controller*)}

EIC = {((*Self*, *CU_START_IN_TOP*), (*Sensor Controller*, *sctrl_start_in*)); ((*Self*, *CU_LIGHT_IN_SU*), (*Sensor Controller*, *sctrl_light_in*))}

EOC= {((*Sensor Controller*, *sctrl_start_out*), (*Self*, *CU_START_OUT_SU*)); ((*Movement Controller*, *mctrl_movel_out*), (*Self*, *CU_MOVEL_OUT_MU*)); ((*Movement Controller*, *mctrl_mover_out*), (*Self*, *CU_MOVER_OUT_MU*))}

IC = {(*Sensor Controller*, *sctrl_mctrl_out*); (*Movement Controller*, *mctrl_sctrl_in*)}

Select = {*Sensor Controller*, *Movement Controller*}.

In this above specification, *X* represents the set of input events, *Y* the set of output events, and *D* each model's component name. *Md* is the set of DEVS basic (atomic or coupled) models, *EIC* the set of external input couplings, *EOC* the set of external output couplings, *IC* the set of internal couplings, and select a tiebreaker function used for simultaneous events.

The DEVS formal specification for the movement controller model shows how we define atomic models:

$M = <X, S, Y, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, ta>$,

where

X = {(*mctrl_sctrl_in*,{*OFF_TRACK,ON_TRACK,STOP_PROC*})}

S = {"*IDLE*", "*WAIT_DATA*", "*PREP_MOVE_FWD*", "*MOVE_FWD*", "*PREP_TURN*", "*TURN_ALPHA*", "*PREP_STOP*", "*MX_STOP*"}

Y = {(*mctrl_mover_out*, {*O_GO_FWD*, *O_GO_REV*, *O_STOP*}); (*mctrl_movel_out*, {*O_GO_FWD*, *O_GO_REV*, *O_STOP*}); (*mctrl_sctrl_out*, {Ø})}

$ta = S \rightarrow R^{+}_{0,\infty}$ and is usually defined as part of δ_{ext} and δ_{int}.

Figure 4 illustrates a DEVS graph representing the movement controller behavior. A state diagram summarizes the behavior of a DEVS atomic component by presenting the states, transitions, inputs, outputs, and state durations graphically. The circles represent states—the double circle is the initial state—and include the state's name and duration. The continuous edges between the states represent the external transitions, which include the names of the input ports, the input value, and any condition on the input (with format "port?value"). The dotted lines represent the internal transitions and the associated outputs (with format "port!value"). Other components of the line-tracking robot are defined in a similar fashion.

Subsequent model-checking involves the following steps[13,14]:

- Replace all models with their RTA-DEVS versions. For our line robot, the RTA-DEVS model is identical to the provided model definition because it complies with RTA-DEVS restrictions.
- Define a clock variable for each atomic RTA-DEVS model.
- Replace every state in the RTA-DEVS model with a corresponding TA one. A location is created for each state with the same name in the TA model.
- Model the RTA-DEVS internal transitions using TA.
- Model the RTA-DEVS external transition using TA.
- Verify properties in the resulting TA (such as absence of deadlocks, liveness, bounded time) via UPPAAL.[13]

After the formal specification and model-checking phases, we use E-CD++ to run simulations, test individual components under different loads, gather results, and derive different test cases. E-CD++ provides a mechanism to program DEVS hierarchical structures. The model definitions and couplings are written in a specific format, and state transitions and output function are overwritten in C++ as part of each model's class definition. Figure 5 shows excerpts of the movement controller's transition and output functions, in accordance with the state diagram in Figure 4.

Lines 25 to 38 show a portion of the internal transition function describing the transition from PREP_MOVE_FWD to MOVE_FWD and PREP_MOVE_FWD to TURN_ALPHA, whereas lines 40 to 48 show a portion of the output function's behavior for a different state. The output function sets the output signal (O_GO_FWD, O_GO_REV, and O_STOP) to send to the motor unit through mctrl_mover_out and mctrl_movel_out.

Using these models, we can test different scenarios early on by using event files that generate events for the input ports. Once the developer is satisfied with the results, the components can be incrementally moved to the target platform. To do this, each driver is associated with specific commands related to the hardware component with which it interacts. Once the model and driver implementation is complete, different tests progressively integrate the hardware components and check the entire system.

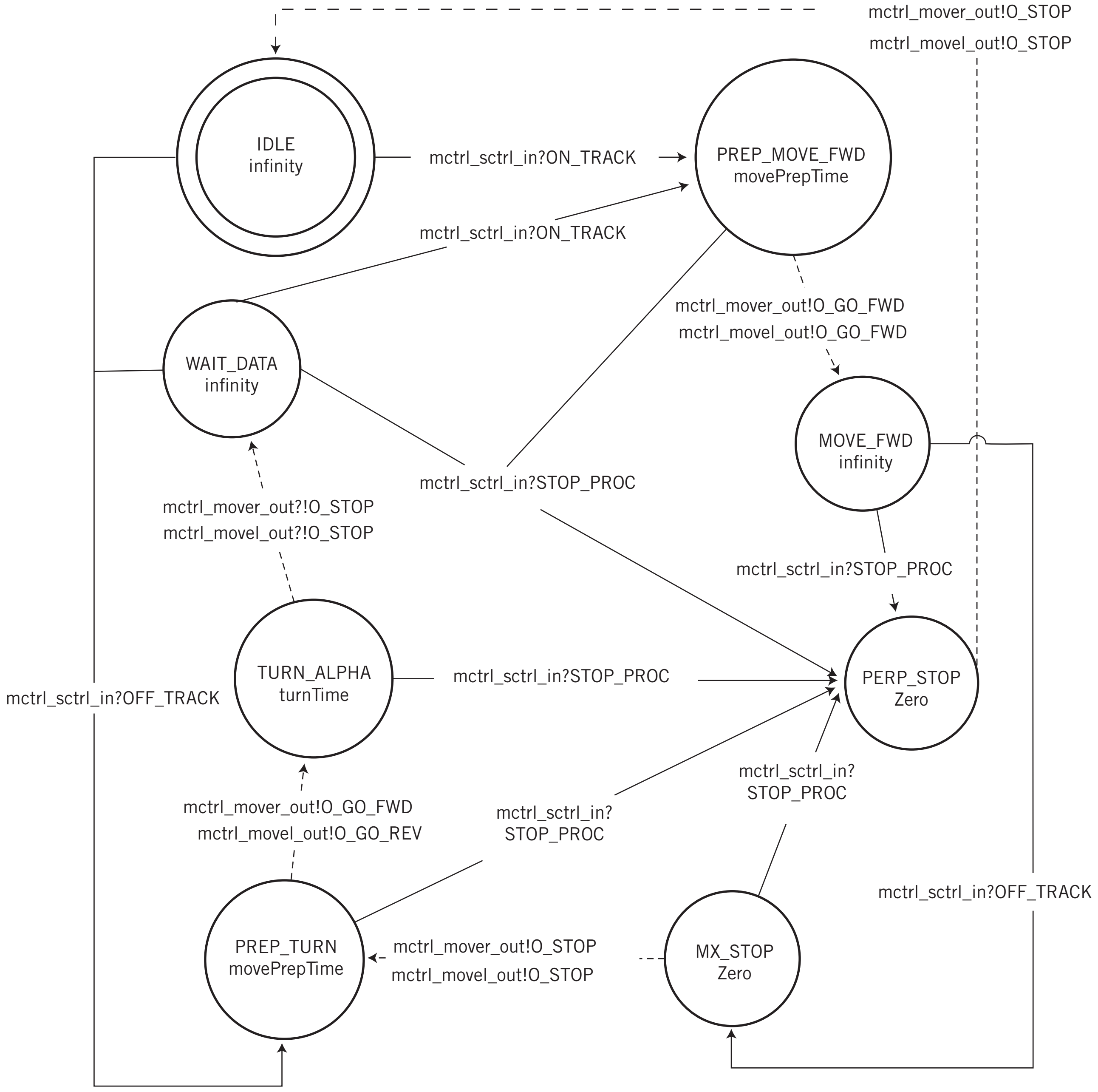


Figure 4. Movement controller state diagram. The circles represent states—the double circle is the initial state—and include the state's name and duration. The continuous edges between the states represent the external transitions, which include the names of the input ports, the input value, and any condition on the input (with format "port?value"). The dotted lines represent the internal transitions and the associated outputs (with format "port!value").

Model Verification and Execution

To test our line-tracking application, we first use a virtual time simulation and perform tests on individual components to ensure that they behave as expected in different environment settings. Then, we perform the tests in real time and integrate hardware components progressively. Table 1 shows a port-mapping table and the description of each value to test the model's behavior.

Table 2 shows an extract of a unit test performed on the movement controller, and Table 3 shows an example of the input events we injected in the system and the resulting outputs.

```
Model &MovementController::externalFunction( const ExternalMessage &msg ){
  if (msg.port() == mctrl_sctrl_in) {   // Sensor Controller signal received
    sctrl_input = static_cast<int>(msg.value());
    if (sctrl_input == ON_TRACK) { // The robot is on the right path
      if (state == WAIT_DATA || state == IDLE) { // Waiting for data or Idle
        state = PREP_MOVE_FWD;         // Prepare to move forward
        holdIn( Atomic::active, movePrepTime );
      }
    }
    else if (sctrl_input == OFF_TRACK) { // The robot is not on the right path
      if (state == MOVE_FWD) {     // if it was moving forward
        state = MX_STOP;       // stop immediately
        holdIn( Atomic::active, ZERO_TIME );
      }
      else if (state == WAIT_DATA || state == IDLE) { // Waiting or idle
        state = PREP_TURN;      // prepare to turn to detect the path
        holdIn( Atomic::active, movePrepTime );
      }
    }
    ...// Remaining case STOP_PROC, go to PREP_STOP in that case
  }
  return *this;
}

Model &MovementController::internalFunction( const InternalMessage & ){
  switch (state){
    case PREP_MOVE_FWD:   // Preparing to move forward
      state = MOVE_FWD;
      passivate();
      break;
    case PREP_TURN:
      state = TURN_ALPHA;
      holdIn( Atomic::active, turnTime );
      break;
    ... // Instructions for remaining states
  }
  return *this;
}

Model &MovementController::outputFunction( const InternalMessage &msg ){
  if(state== MX_STOP || state==TURN_ALPHA || state== PREP_STOP){
    sendOutput( msg.time(), mctrl_mover_out, O_STOP) ;
    sendOutput( msg.time(), mctrl_movel_out, O_STOP) ;
  }
  ...// Remaining cases: state==PREP_TURN, send O_GO_FWD and O_GO_REV
    //         state=PREP_MOVE_FWD, send O_GO_FWD to both motors
  return *this ;
}
```

Figure 5. Excerpts of the movement controller's transition and output functions from Figure 4.

Table 1. Port mapping.

Port name	Port value	Hardware command	Description
START_IN	10	START	Manual start command
	11	STOP	Manual stop command
LIGHT_IN	0	BRIGHT	No line detected
	1	DARK	Line detected
	2	ALL_DARK	Destination reached
MOVER_OUT/ MOVEL_OUT	0 1 2	STOP FORWARD REVERSE	Stops the motor Spins clockwise Spins anticlockwise

Table 2. Movement controller's simulated input events and resulting output.

Input	Output	Comments
00:00:01:030 sctrl_mctrl_out ON_TRACK	00:00:01:080 mover_out 1 00:00:01:080 movel_out 1	ON_TRACK received Moving forward
00:00:02:000 sctrl_mctrl_out ON_TRACK	–	No change in commands
00:00:02:050 sctrl_mctrl_out OFF_TRACK	00:00:02:050 mover_out 0 00:00:02:050 movel_out 0	OFF_TRACK received Stop motors immediately Turn Stop motors after turn
	00:00:02:100 mover_out 1 00:00:02:100 movel_out 2	
	00:00:03:100 mover_out 0 00:00:03:100 movel_out 0	
00:01:03:50 sctrl_mctrl_out STOP_PROC	00:01:03:050 mover_out 0 00:01:03:050 movel_out 0	STOP_PROC received Stop immediately

After one second, an input to the START_IN input port starts the system. Then, at 2 seconds, a value of 1, meaning the line is detected, is sent through the LIGHT_IN input port. To illustrate situations when the robot gets off track, a value of 0 is sent through the LIGHT_IN port. The system is then manually stopped by sending 11 through the START_IN port. Different values are sent through the LIGHT_IN port to test how the system behaves after a manual stop.

The resulting behavior is similar to the one defined in the controller models. Indeed, when the robot goes off track and doesn't detect a line, it stops, turns counterclockwise slightly, and then tries to detect a trail again. If a line is detected, the robot moves forward again; otherwise, it continues to turn until it finds a path to follow. The destination is considered to be a wide dark ground surface. Once this surface is detected, the robot will stop and go into an idle state.

When porting the same models to be executed in real time on an ARM microcontroller, we observed the same behavior. After running various scenarios to verify the model behavior on the board, we mapped the driver interfaces with the robot sensors and actuators. Figure 6 shows the robot shield and the board used for this application. A button and a reflectance sensor are connected to the shield.

The START_IN driver is attached to the button for starting/stopping the robot and acts as an active device in this case. The LIGHT_IN driver is associated with the reflectance sensor for sensing surface brightness and acts as a passive device because polling is needed to collect the sensor values. The output drivers MOVER_OUT and MOVEL_OUT are connected to two servomotors. The same models can be reused on another hardware platform without any modification. We performed the tests on both the Lego (http://youtu.be/mTtlSV7WbuI) and ARM robots (http://youtu.be/X2itlznkoVw)—the robot followed the line as expected.

Table 3. Simulated input events and resulting output.

Input	Output	Comments
00:00:01:000 START_IN 10	–	System START, no output
00:00:02:000 LIGHT_IN 1	00:00:02:200:000 mover_out 1	Robot moves forward
	00:00:02:200:000 movel_out 1	
00:00:02:500 LIGHT_IN 0	00:00:02:600:000 mover_out 0	OFF_TRACK signal, robot stops
	00:00:02:600:000 movel_out 0	
	00:00:02:700:000 mover_out 1	Robot turns
	00:00:02:700:000 movel_out 2	
00:00:02:700 START_IN 11	00:00:02:700:000 mover_out 0	Manual system STOP, turn interrupted
	00:00:02:700:000 movel_out 0	
00:00:03:000 LIGHT_IN 1	–	Ignored – STOPPED
00:00:03:500 LIGHT_IN 0	–	Ignored – STOPPED

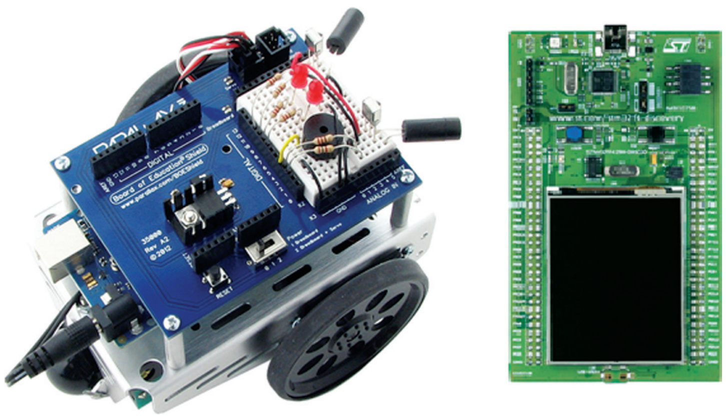


Figure 6. Hardware platform (Parallax robot shield + STM32F429 discovery board). The START_IN driver is attached to the button for starting/stopping the robot and acts as an active device. The output drivers MOVER_OUT and MOVEL_OUT are connected to two servomotors.

The increasing demand for new features and quality combined with decreasing budgets and time to market pose great challenges to embedded system designers. Building these products is technologically and economically feasible, but software design, testing, and integration are still the most expensive tasks. Our methodology addresses these challenges, but we acknowledge there is much to be done. ■

References

1. F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, Wiley & Sons, 2001.
2. B. Zeigler, H. Praehofer, and T. Kim, *Theory of Modeling and Simulation*, Academic Press, 2000.
3. H. Dongping and H. Sarjoughian, "Software and Simulation Modeling for Real-Time Software-Intensive Systems," *Proc. 8th IEEE Int'l Symp. Distributed Simulation and Real-Time Applications*, 2004, pp. 192–203.
4. X. Hu and B. Zeigler, "Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example," *J. Intelligent and Robotic Systems*, vol. 39, no. 1, 2004, pp. 71–87.
5. H. Saadawi and G. Wainer, "From DEVS to RTA-DEVS," *Proc. 14th IEEE Int'l Symp. Distributed Simulation and Real Time Applications*, 2010, pp. 207–210.
6. H. Saadawi and G. Wainer, "Principles of DEVS Models Verification for Real-Time Embedded Applications," *Real-Time Simulation Technologies: Principles, Methodologies, and Applications*, CRC Press, 2011, pp. 63–96.
7. G. Wainer, *Discrete-Event Modeling and Simulation*, CRC Press, 2009.
8. E. Kofman and S. Junco, "Quantized-State Systems: A DEVS Approach for Continuous System Simulation," *Trans. Soc. Computer Simulation*, vol. 19, no. 3, 2001, pp. 123–132.
9. G. Wainer, "Cell-DEVS/GDEVS for Complex Continuous Systems," *Simulation*, vol. 81, no. 2, 2005, pp. 137–151.
10. M. Sipper, "The Emergence of Cellular Computing," *Computer*, vol. 32, no. 7, 1999, pp. 18–26.

11. R. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM*, vol. 33, no. 10, 1990, pp. 30–53.
12. S. Jafer, Q. Liu, and G. Wainer, "Synchronization Methods in Parallel and Distributed Discrete-Event Simulation," *Simulation Modelling Practice and Theory*, vol. 30, 2013, pp. 54–73.
13. G. Behrmann, A. David, and K. Larsen, "A Tutorial on Uppaal," *Formal Methods for the Design of Real-Time Systems*, 1st ed., M. Bernardo and F. Corradini, eds., Springer Berlin Heidelberg, 2004, pp. 200–236.
14. H. Saadawi and G. Wainer, "Principles of Discrete Event System Specification Model Verification," *Simulation*, vol. 89, no. 1, 2011, pp. 41–67.

Daniella Niyonkuru is an MS student in the Department of Systems and Computer Engineering at Carleton University. Her research interests include embedded software development and modeling, and simulation-driven development. Niyonkuru has a BE in computer systems from Université du Québec en Outaouais. Contact her at daniella.niyonkuru@carleton.ca.

Gabriel A. Wainer is a professor in the Department of Systems and Computer Engineering at Carleton University. His research interests are in modeling, simulation, and real-time embedded systems. Wainer has a PhD in computer science from the Université d'Aix-Marseille III, France. Contact him at gwainer@sce.carleton.ca or via http://cell-devs.sce.carleton.ca/.

stay connected.
Keep up with the latest IEEE Computer Society publications and activities wherever you are.
IEEE computer society