A Kernel for Embedded Systems Development and Simulation using the Boost Library

Daniella Niyonkuru **Gabriel Wainer**
Dept. of Systems and Computer Engineering, Carleton University
1125 Colonel By Dr., Ottawa, ON, Canada K1S 5B6
{daniella.niyonkuru, gabriel.wainer}@carleton.ca

ABSTRACT
Model-Driven Development is a promising solution to handle the complexity of embedded systems development. This approach uses models as principal artifacts throughout the entire development cycle. In this paper, we present a bare-metal kernel that executes DEVS models on hardware. Our solution uses the Boost library, and can be interfaced with diverse hardware libraries. We detail the internal architecture as well as the DEVS execution mechanism at the core of the kernel. We also demonstrate the usability with a case study that shows how models become the real controllers.

1. INTRODUCTION
The embedded systems industry is facing an increasing complexity of the systems' functionality and shorter time-to-market expectations. The demand of more contents and new intelligent capabilities is a mismatch with the request for a reduced product development time, increased robustness, and future extensibility. Moreover, traditional design methods where systems are designed directly at the low hardware and software levels are fast becoming infeasible due to the increasing complexity and market demands.

Model-based techniques, where models drive the development, are a promising solution to lessen the productivity gap and deal with the current challenges [1]. We are interested in investigating a model-driven approach called the Discrete Event Methodology for Embedded Systems (DEMES). In DEMES, DEVS (Discrete Event System Specification) [2] models are used consistently through the development cycle until they are deployed on the target hardware. One of the key aspects of this process is a real-time (RT) executive that runs the models on the chosen hardware, which must be efficient and have small memory footprint. In this context we introduce a new RT executive named Embedded-CDBoost (*E-CDBoost*).

Here, we present an overview of the DEMES development cycle and then explain the design of E-CDBoost. Finally, we will illustrate its application with a case study and compare the new kernel's performance against an existing

SpringSim-TMS/DEVS 2016 April 3-6 Pasadena, CA, USA

2. BACKGROUND
E-CDBoost is based on DEVS, which is suitable for RT embedded systems as it provides a rich structural representation of components, and formal means for explicitly specifying their timing, which is central for RT systems.

DEMES uses formal models in order to analyze real-time embedded systems and study their interaction with the physical environment while enabling original models to be part of the final product. This is done by replacing models incrementally with hardware surrogates and new software components without altering the original models. The transition can be done in incremental steps; models are incorporated in the target environment after extensive testing, and reused throughout the entire development process.

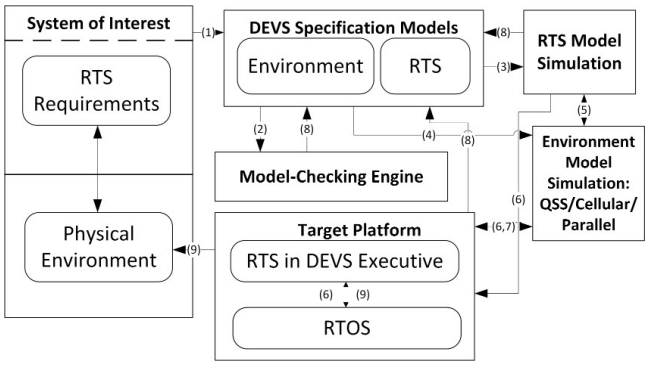

Figure 1. Discrete-Event Methodology for Embedded Systems [3]

The DEMES development cycle involves different steps. The system of interest is first defined in terms of its requirements and the relation with the physical environment. These latter are then formally specified with DEVS. DEVS models can be used to perform formal verification and run simulations under different environmental settings. Once the models have been tested and verified, they are deployed onto the target platform where a RT executive runs them.

The RT executive is based on the DEVS execution semantics. In the model specification phase, the formalism decomposes complex designs into basic (behavioral) models called atomic, and composite (structural) models called coupled [8]. It follows a precise rule set to define state changes of the modeled systems depending on input events or time delay triggers. With its abstract simulator, two kinds of components or processors are defined: *Simulators* (in charge of atomic models), and *Coordinators*

(in charge of coupled models). Simulators are the engines that invoke the model transition functions (δint, δext, δconf, ta, λ) and Coordinators in charge of event routing and hierarchical scheduling. Parent and children models communicate via message passing mechanism in order to render DEVS model behavior. The abstract simulator also provides a set of algorithms for coordinators and simulators that specify how each engine reacts upon the reception of each message.

DEVS tools use the previously described execution mechanisms by implementing message passing and algorithms provided with the abstract simulator. Some researchers have particularly looked at low-level applications. These include DEVSJAVA [5], a Java-based DEVS simulator that supports high-level modeling; RTDEVS/CORBA [6] [12], a DEVS implementation based on real time CORBA; and PowerDEVS [13], a tool for hybrid system modeling and RT simulation. Most of these solutions, however, use an operating system. For instance, E-CD++ uses a variant of the Linux kernel [10]. In [14] the authors use a TINI chip, which requires Java Virtual Memory and Java class libraries on the chip. PowerDEVS uses Linux RTAI [13]. Instead, we built a bare-metal version of E-CD++ that can run on microcontrollers with limited memory resources [15]. We needed to remove existing OS dependencies and adding new components to enable standalone execution.

E-CDBoost, the new bare-metal RT executive, is extended from CDBoost [16], a DEVS simulator built around a sequential PDEVS architecture [16]. Figure 2 shows the architecture overview. As with DEVS, it separates the model construction logic from the simulation mechanism.

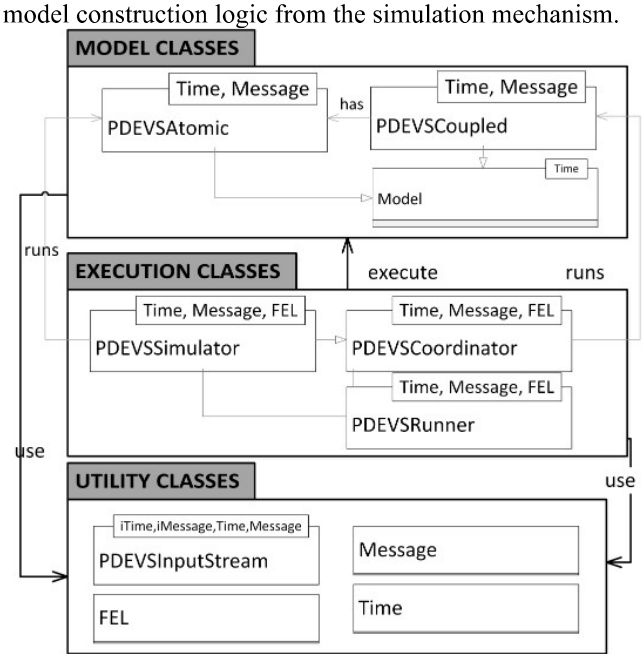

Figure 2. CDBoost, Software Components Overview [13]

Model classes provide the former while execution classes implement the latter. Utility classes provide useful functions such as time classes, message classes, input stream for external events and a future event list. Model classes contain three main classes: *Model* that offers a common interface to atomic and coupled models, *PDEVSAtomic* can be extended to implement user defined atomic models, and *PDEVSCoupled* provides an interface to specify the structure of a model. Execution classes, on the other hand, group *PDEVSSimulator* to define atomic models, *PDEVSCoordinator* to execute coupled models, and *PDEVSRunner,* similar to the Root Coordinator.

CDBoost replaces top-down messages by function calls and bottom-up messages with returns. In other DEVS simulators, nodes in the Processor hierarchy communicate by sending messages and executing actions locally; in CDBoost, functions of lower nodes are called and values returned (i.e. *advance_simulation*() and *collect_outputs()*).

3. EMBEDDED CD-BOOST

To allow model execution directly on hardware, we used a DEVS bare-metal kernel that includes the E-CDBoost RT executive. Other components of the kernel include a microkernel - handling system calls related to file, memory and input/output management – and a hardware abstract layer that interfaces with multiple hardware peripheral libraries – as show in Figure 3. We will particularly focus on the RT executive derived from CDBoost.

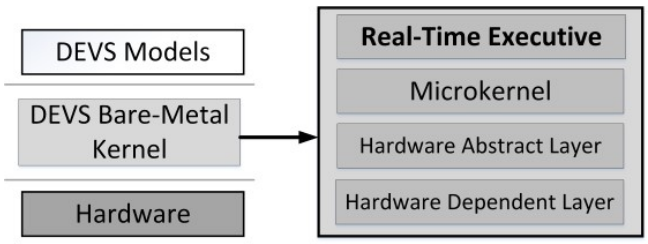

Figure 3. DEVS Model Execution on Bare-Metal

Before deployment on the target platform, modelers implement their models using an integrated development environment (in our case, Eclipse), which is also used for simulation and debugging. The resulting model files are linked with RT executive components, microkernel elements and peripheral libraries. The resulting firmware is then deployed onto the microcontroller where models act as controllers. The RT executive is responsible of routing hardware events from the environment to the models and vice-versa.

3.1. Architecture Overview

Embedded CDBoost (E-CDBoost) is designed to execute models on embedded hardware; this needs real-time execution and interaction with the environment. E-CDBoost can read inputs from hardware components (sensors, timers, etc.), and actuate motors, valves, gears etc. It supports the integration of both simulated and real components for hardware-software co-design. Since E-CDBoost is designed

to execute in RT, it includes a wall-clock time (with a microsecond precision) interfaced with a hardware timer.

E-CDBoost adds a *Port* component to the modelling subsystem in Figure 2, a *Driver* and a new *Runner* (ERunner) to interface model implementation and hardware platform. These elements are shown in Figure 4, and they were added in order to allow communication with the environment, retrieve values from sensors and send commands to actuators.

The physical time management is handled by a special Time class. The message structure is also adapted to the RT environment and carries port and value information.

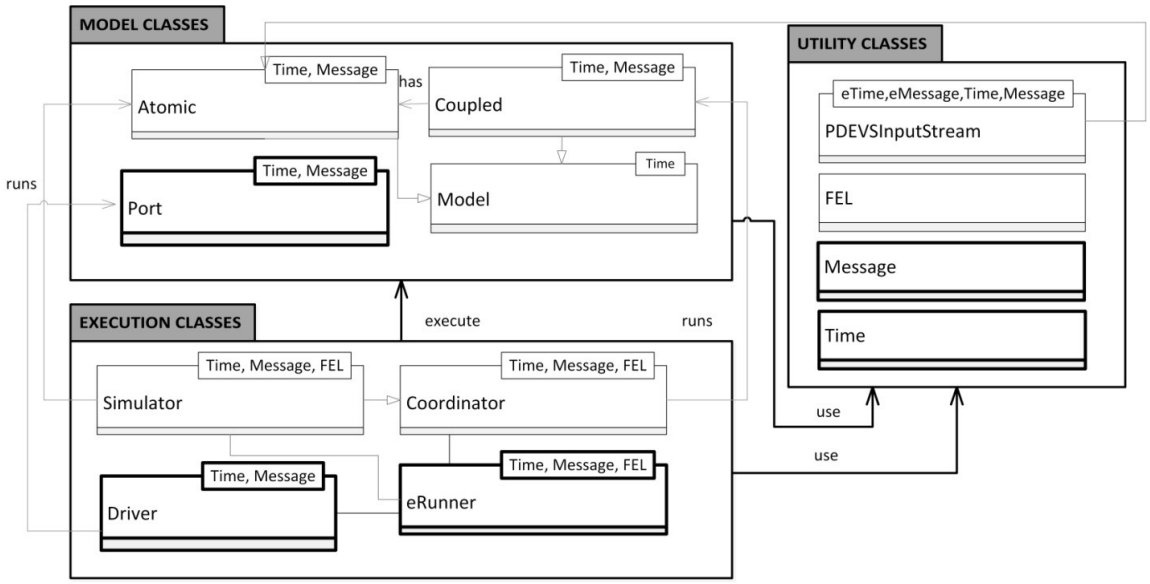

Figure 4. E-CDBoost Software Architecture Overview.

3.2. Subsystems Overview

We preserved the construction and execution in DEVS, using three subsystems or categories of classes, as described in Section 2.4. Ancillary classes contain structures and other components needed for the execution process. With the above additions, hardware-in-the-loop simulation is now possible. If the hardware components (e.g. sensors) are not available, a model can be used and tested along with the available components. This allows us to combine both hardware and simulated units and start testing early in the development process. When all the components are available, control models interact directly with the hardware devices through the defined ports.

4. SUBSYSTEMS IMPLEMENTATION

Coupled models are defined using the *PDEVSCoupled* class. This class constructor receives four parameters:

- the list of pointers of the components;
- the External Input Couplings (EIC) pointers list;
- the Internal Couplings (IC) pointers list;
- the External Output Couplings (EOC) pointers list.

Both *PDEVSAtomic* and *PDEVSCoupled* inherit the model class that allows coupled and atomic models to be connected easily through couplings that can be debugged with ease since they share a common model interface.

PDEVSAtomic and *PDEVSCoupled* are common to both CDBoost and E-CDBoost. E-CDBoost adds "Port" to its modeling subsystem. This represents the logical connection between models and hardware devices. The user must provide ports implementation by extending a Port base class, and specifying a "*pDriver*" or port driver function to translate model output values to specific hardware components.

When defining a top port, the user specifies the name, related EIC/EOC, and a *polling period*. This can be useful to customize the polling frequency of input devices as sensors.

The link between DEVS logic and the hardware peripheral libraries is established by *pDriver*, which either receives a value to be translated onto commands, or it returns a value depending on the state of input devices. For an input port,

pDriver could provide which GPIO (General Purpose Input Output) pin to read, and set the port value. For an output port, it receives a value that is translated into actions.

Execution classes, illustrated in Figure 4, implement the abstract simulator algorithms and execute models. The *PDEVSCoordinator* class, in charge of managing coupled models, requires three template parameters: *Time*, *Message* and *Future Event List (FEL)*. These parameters will be detailed in the ancillary subsystem with *utility* classes.

The Processor hierarchy is constructed by the invocation of a constructor. Constructing coordinator objects requires the coupled model components to be extracted and embedded in the coordinator. For instance, when the Coordinator is built, all the children are constructed, and the couplings between components are saved. The algorithms described previously – *collect_outputs* and *advance_simulation* (here *advance_execution*) are implemented in these classes.

The *PDEVSSimulator* class implements the simulator's algorithms introduced in section 3. This class calls the state transition functions and returns the outputs of the atomic models to their Coordinators. E-CDBoost uses a flat coordinator and adds a global driver ("Driver") that manages ports. Simulators are linked to a top flat coordinator.

Root Coordinator

It is created and driven by the *eRunner* class, manages the global execution and defines the end time of the simulation.

```
run():
while curentTime < stopTime
  wait for is signals from environment
or internal time out(tN)
   if external event then
        Message in = DX(is)
        topCoordinator.postEvent(in)
        topCoordinator.advance_execution()
    else if internal time out then
        topCoordinator.collect_outputs()
        if output messages out received then
            os = DY(out)
            send os signal to hardware
        end if
        topCoordinator.advance_execution()
    end if
    tN = topCoordinator.next()
end while
```

The default stop time is infinity as in typical embedded systems a program is set to run forever. It waits for an internal or external event in order to advance execution. In the first case, outputs are collected and *advance_execution*. When an external event occurs, the event value is added to the top coordinator and *advance_execution* is called to process it. When the runner receives an output message, it is processed by the driver, and a corresponding value is sent to the port. Input event values are retrieved from the global driver.

Global Driver Object

The Driver is responsible for initializing hardware, retrieving inputs from hardware components connected to input ports (calling *pDriver*), and sending commands to hardware components connected to the output ports.

When a signal is detected on an input port, a message is generated and added to the top coordinator inbox. The input event retrieving mechanism is based on polling interrupts. The user may also choose to use interrupts for signal detection. One of the advantages of our approach (especially for experienced embedded systems developers), is that they can use hardware or software interrupts to detect changes on the hardware components directly and generate input messages. Indeed, specific hardware interrupts associated with each hardware device can be used to signal an input event while software interrupts can be programmed based on a division of the base clock to provide periodic polling. Interrupt service routines are then set to post a port value that is then used by the port driver to generate a PDEVS message.

In the case of output message, the driver will call the related output port pDriver - in charge of converting the received message into commands - with the received output data.

The *utility* classes provide essential data structures in order to run the model. The first class in the utility category is called *Message*. Boost::any is used by default in CDBoost, as it allows the exchange of any type of messages in our models. In E-CDBoost, we have defined a special message type that includes time, port, and value parameters. The *Time* component is associated with the physical time and provides a RT clock with microsecond precision. It is interfaced with a 32-bit hardware timer. The Future Event List (FEL) is provided as part of the utility classes. Using an effective FEL is essential in order to achieve good performance. For the FEL type, any structure that matches the priority queue signature is allowed. Consequently, the user can define personalized schedulers and increase performance if needed. The default FEL we provided is a standard priority queue. This is part of the C++ language and is suited to store and retrieve timed events.

4.1. Execution on the target platform

E-CDBoost runs on top of the microkernel introduced in section 3. This latter handles system calls and provides requested services to the RT executive. The RT executive communicates with hardware via ports and drivers.

A hardware abstraction layer that invokes MBED - a development platform for ARM microcontrollers and connects the application with the underlying hardware - is used to streamline the development and ease applications porting.

5. A LINE TRACKING ROBOT

We have followed the DEMES approach to build several applications, and executed them on the target platform using the new kernel. In this section, we will particularly focus on one application and present how it evolved

progressively from its system of interest definition, to formal model, to the real system. We will see how model-driven practically work and how we can construct a model of a system that we can then transform into the real thing.

5.1. System Description

The first step is to define a system of interest. Ours is a line-tracking robot designed to follow a path identified by a black line and get back on track if the trail is not detected. The system requirements are as follows: the robot shall be equipped with a light sensor that faces the ground and measures the amount of light reflected off a small ground surface. The controller should consider a medium percentage of reflected light as a detected path and initiate the robot to move forward. When the robot goes off track, i.e. does not sense a path trail; it stops, turns slightly, and then tries to detect a trail again. If a path is detected, the robot moves forward again; otherwise, it continues to turn until it finds a path to follow. The robot should also be able to receive manual signals to start and stop.

Model Components

Once the system of interest is defined, the following step is to model the system using DEVS. This formalism, as introduced in section 2, decomposes complex system designs into basic/behavioral models (atomic models) and composite/structural models (coupled models). We take a top down approach and first define the structure of the line tracking robot system. Multiple iterations are usually required to capture the requirements into an appropriate hierarchical structure. Note that we use the same example and hierarchy as in [15] for comparison purposes.

The system is partitioned into three main units: a Sensor Unit, a Control unit, and a Movement Unit. To communicate with the environment, we use two input ports (LIGHT_IN and START_IN), and two output ports (MOVEL_OUT and MOVER_OUT). LIGHT_IN is the input port through which reflected light is measured. START _IN is for the manual start/stop commands. The output ports are for the robot's left and right motors movements.

In terms of components, the sensor unit contains input devices. In this case, it contains an atomic model (light sensor), which reads the amount of light reflected and transmits those readings to the control unit. This latter has a sensor controller and the movement controller. The sensor controller activates or stops the light sensor, receives the sensor readings, and sends messages to the movement controller, specifying whether the robot is on track, off track, or has reached the destination. When the robot arrives at its destination—i.e. the light sensor reads an all-dark surface—the sensor controller sends a "stop reading" command to the light sensor and a stop signal to the movement controller. The movement controller also receives on/off track and stop signals from the sensor controller, and it sends appropriate commands to the motors. The movement unit is made of motor left and motor right. It groups the robot's actuators that move in response to commands received from the control unit. The motor models control the robot movements: they can spin clockwise, anticlockwise, or stop according to the signals they receive from the control unit.

These models can be formally specified and used for model-checking,or formal verification. The DEVS model specification is also preserved as much as possible throughout the development cycle.

Model Specification

The specification of the control unit is shown below as an example. As mentioned earlier, the control unit has two atomic models, the sensor and movement controllers. The control unit can be formally defined as:

$$CM = < X, Y, D, \{Md\}, EIC, EOC, IC >,$$

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 Ctl, Movement Ctl}.
Md = {M(sensor Ctl), M(movement Ctl)}
EIC = {((Self, CU_START_IN_TOP), (Sensor Ctl, sctrl_start_in));((Self, CU_LIGHT_IN_SU), (Sensor Ctl, sctrl_light_in))}
EOC={((Sensor Ctl, sctrl_start_out), (Self, CU_ START_OUT_SU)); ((Movement Ctl, mctrl_movel _out), (Self, CU_MOVEL_OUT_MU)); ((Movement Ctl, mctrl_mover_out), (Self, CU_MOVER_OUT_ MU))}
IC = { (Sensor Ctl, sctrl_mctrl_out); (Movement Ctl, mctrl_sctrl_in) }

Figure 5 illustrates a DEVS Graph representing the sensor controller's behavior.

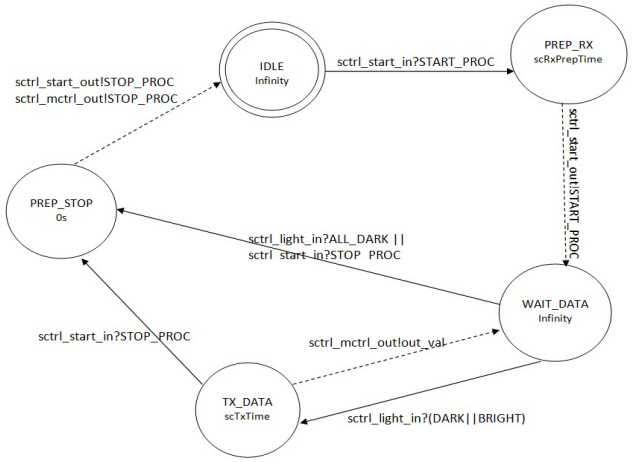

Figure 5. Sensor Controller State Diagram

The Sensor Controller is IDLE until a *start* command is issued. Then, an external transition is triggered and the Sensor Controller state changes to PREP_RX. At this point, it waits for *ta=scRxPrepTime*, after which a 'start' output is

sent to the Light Sensor and an internal transition changes state to WAIT_DATA. It waits in this state until it receives a signal from the Light Sensor. If the signal indicates that the robot reached the destination (ALL_DARK), the external transition causes a switch to PREP_STOP, where it will immediately send a stop signal to the Light Sensor and the Movement Controller, and it will transition back to IDLE. However, if the signal is different, the Sensor Controller will go to TX_DATA, will wait for *ta=scTxTime*, after which it will send an output to the Movement Controller indicating whether the robot is on track or not. If the Sensor Controller receives a manual stop signal (STOP_PROC), it will transition to the PREP_STOP to stop all activities.

5.2. Implementation with E-CDBoost

The user implements atomic models in E-CDBoost by extending a basic model class and providing state transition and output functions. This case study was built in E-CD++ [15]; here, we show the implementation differences. The code below shows an example for the sensor controller functions. We can see that it includes the state transition and output functions that corresponds to the original DEVS specification/graph. In this way, it is similar to [15] except that the time advance is clearly separated. The message structure is constructed using the port and the value to be sent. This structure is specific to E-CDBoost and is not available per default in CDBoost. The TIME parameter returned by the time advance function is defined using real time units, an addition of E-CDBoost too.

```
void internal() noexcept {
    switch (_state){
        case PREP_STOP:
            _state = IDLE;   _next = infinity;
            break;
        case PREP_RX:
        case TX_DATA:
            _state = WAIT_DATA; _next = infinity;
            break;       }
}

/* @return Time until next internal event.  */
TIME advance() const noexcept { return _next; }

/* @return a bag of output messages */
std::vector<MSG> out() const noexcept {
    //…
 switch (_state){
   case PREP_STOP:
   //Send stop through sctrl_start_out and mctrl
   _outputMessage1 = MSG(portName[sctrl_start_out],
               STOP_PROC);
   _outputMessage2 = MSG(portName[sctrl_mctrl_out],
               STOP_PROC);
   std::vector<MSG>{_outputMessage1, _outputMessage2};

   case PREP_RX: //Send Start through sctrl_start_out
     _outputMessage1 = MSG(portName[sctrl_start_out],
                 START_PROC);
    return std::vector<MSG>{_outputMessage1};

  case TX_DATA: {
  //Send on/off track signals sctrl_mctrl_out
     int output_val;
  if(sensor_input == DARK) output_val = ON_TRACK;
      else if (sensor_input == BRIGHT)
              output_val = OFF_TRACK;

  _outputMessage1 = MSG(portName[sctrl_mctrl_out],
               output_val);
  return std::vector<MSG>{_outputMessage1};
       }
    };
   return std::vector<MSG>{}; //Default: empty output
}
```

To implement coupled models, input, internal and output links have to be provided. The following snippet shows how the control unit model is described in E-CDBoost.

```
1.  // Atomic models definition
2.  auto sctrl = make_atomic_ptr
    <SensorController<Time, Message>>();
3.  auto mctrl = make_atomic_ptr
    <MovementController<Time, Message>>();
4.  //Coupled model definition
5.  shared_ptr<flattened_coupled<Time, Message>>
    ControlUnit( new flattened_coupled<Time,
    Message>{{sctrl,mctrl}, {sctrl},
    {{sctrl,mctrl}}, {mctrl}});
```

The sensor controller (sctrl at line 2) and movement controller (mctrl at line 3) are the two components of the control unit. The model is created on line 5 by respectively providing its components ({sctrl,mctrl}), then its EIC (signals from hardware components; sctrl is connected to the light sensor and push button), its IC (sctrl is connected to mctrl internally), and finally its EOC (components sending output signal to hardware: mctrl to the two motors). One of the advantages of this approach is that no file needs to be embedded onto the target platform or converted beforehand. It also offers a lightweight mechanism for specifying links.

Once satisfied with the simulation results, hardware components and DEVS controller are integrated. To interface models with hardware components, EIC and EOC components are linked to top ports, as follows:

```
1.  // Input ports
2.  auto start = make_port_ptr<START_IN
        <Time, Message>>();
3.  auto light = make_port_ptr<LIGHT_IN
        <Time, Message>>();
4.  // Output ports
5.  auto motorleft = make_port_ptr<MOVEL_OUT
        <Time, Message>>();
6.  auto motorright =
    make_port_ptr<MOVER_OUT<Time, Message>>();
7.  // Execution parameter definition
8.  erunner<Time, Message> root{ControlUnit,
    {{start,sctrl},{light,sctrl}} ,
    {{motorleft,mctrl},{motorright,mctrl}}
    };//link top ports to EIC and EOC components
```

Lines 2 and 3 create the two input ports respectively connected to the start button and the light sensor. Line 5 and 6 show the two output ports linked to the motors. Links between ports and the model they are connected to are

passed along with the top model to the *erunner* (defined in section 4) that executes models on the target platform.

For hardware integration, we use a Seeed Studio Shield bot and a Nucleo development board. One of the onboard reflectance sensors is used as the input for our light readings, a push button on the Nucleo and the two motors of the Seeed Shield Bot to move the robot.

```
template<class TIME, class MSG>
class LIGHT_IN : public port<TIME, MSG>  {
public:
    /* @param n Name assigned to the port.
       @param polling Polling for the port  */

  explicit LIGHT_IN(const std::string &n =
      "light_in", const TIME &polling =
        TIME(0,0,0,200)) noexcept : port<TIME,
          MSG>(n,polling) {}
  bool pDriver(Value &v) const noexcept;        };
```

Top ports connected to hardware sensors/actuators have to be specified to interface the model with the previous hardware components. These ports are specified as extension of a basic port class. The LIGHT_IN port is derived from the port class and provides a default polling time (200 ms here) when interrupts are not used by the user. In its *pDriver* implementation (shown below), we call a function of the Seeed Shield Bot MBED library that returns the value of the onboard sensor used to track the line.

```
template<class TIME, class MSG>
bool LIGHT_IN<TIME, MSG>::pDriver(Value &v) const
noexcept {    v = bot.getCentreSensor();
              return true;                    }
```

Bot is defined during the hardware initialization process and it contains the hardware pins connected to the hardware bot. In this case, the centre sensor is connected to A2.

```
SeeedStudioShieldBot bot(
    D8, D9, D11,              // Left motor pins
    D12, D10, D13,            // Right motor pins
    A0, A1, A2, A3, D4        // Sensors pins       );
```

6. RESULTS

We will illustrate the execution mechanism using trace logs collected during the execution of the line tracking robot. It illustrates the advance_simulation/execution() and collect_ouputs() function calls explained earlier. The flat coordinator forwards the function call to the appropriate simulator which, in turn returns outputs or calls its state transition functions. Two examples are provided to illustrate internal execution mechanism are shown below.

```
DRIVER: INPUT MESSAGE     Time: 00:00:02:517:459
 Port: start_in Value: 10
 - advance_execution()::flattop
 - advance_execution()::sctrl
            model->external() model->advance(): 00:00:00:040:000
 - collect_outputs()::flattop
 - advance_execution()::flattop
 - collect_outputs()::sctrl            model->out()
 - advance_execution()::sctrl          model->internal() model->advance(): ...
 - advance_execution()::mctrl          model->external() model->advance(): ...
DRIVER: INPUT MESSAGE  Time: 00:00:02:600:697
 Port: light_in Value: 1
 - advance_execution()::flattop
 - advance_execution()::sctrl
            model->external() model->advance(): 00:00:00:040:000
 - collect_outputs()::flattop
 - advance_execution()::flattop
 - collect_outputs()::sctrl            model->out()
 - advance_execution()::sctrl          model->internal() model->advance(): ...
 - advance_execution()::mctrl
            model->external() model->advance(): 00:00:00:040:000
 - collect_outputs()::flattop
 - collect_outputs()::mctrl            model->out()
DRIVER: OUTPUT MESSAGE  Time: 00:00:02:680:850
 Port: motor1 Value: 1
DRIVER: OUTPUT MESSAGE   Time: 00:00:02:680:834
 Port: motor2 Value: 1
```

The listing above shows the sequence that follows a start button press at time 00:00:02:517:459. The driver constructs an input message that triggers the call of the external function of the sensor controller model. An input message indicating a line detection is then sent by the driver and causes the sensor and movement controller external functions to be called. Two outputs are generated, commanding the motors to go forward (Value 1 sent to both motors).

The listing below shows the case corresponding to a manual stop that causes stop commands (0 sent to motor1 and motor2) to be sent to the motors.

```
DRIVER: INPUT MESSAGE  Time: 00:02:10:403:002
 Port: start_in Value: 11
 - advance_execution()::flattop
 - advance_execution()::sctrl
            model->external() model->advance(): 00:00:00:000:000
 - collect_outputs()::flattop
 - advance_execution()::flattop
 - collect_outputs()::sctrl            model->out()
 - advance_execution()::sctrl          model->internal() model->advance(): ...
 - advance_execution()::mctrl
            model->external() model->advance(): 00:00:00:000:000
 - collect_outputs()::flattop
 - collect_outputs()::mctrl            model->out()
DRIVER: OUTPUT MESSAGE  Time: 00:02:10:403:559
 Port: motor1 Value: 0
DRIVER: OUTPUT MESSAGE  Time: 00:02:10:403:543
 Port: motor2 Value: 0
```

Once the tests are done, the controller model is deployed onto the Nucleo board to autonomously control the robot. A video showing the result on the target platform is available here [18].

Two of the desired outcomes of E-CDBoost were a smaller kernel footprint and a decreased overhead. In terms of code size, some kernel design decisions, such as the inclusion of the nanolib – an optimized library for microcontrollers -, allowed us to reduce the code size by more than 50%. We also compared the code size of E-CD++ and E-CDBoost. The latter is smaller. For the line tracking robot application, E-CDBoost occupies 131 KB of flash memory and 448 bytes of data memory while the E-CD++ takes 240 KB of flash memory and 608 bytes of data memory.

We compared the performance of both techniques for this line tracking robot application. We particularly measured the time it takes for an external event to trigger the external function of a model, i.e. the time it takes from the root to the simulator (EXT: Root to Simulator in Table 1). We also assessed the time it takes from the external function to return control to the root (EXT: Simulator to Root in Table 1). The other aspect that we examined was the output collection, specifically the time it takes from the root collect outputs command to the output function call (OUT: Root to Simulator) and for the outputs to be received by the driver object (OUT: Simulator to Root). The following table summarizes the results.

	E-CD++	**E-CDBoost**
EXT: Root to Simulator	155 us	53 us
EXT: Simulator to Root	159 us	43 us
OUT: Root to Simulator	68 us	25 us
OUT: Simulator to Root	97 us	31 us

Table 1. Overhead Evaluation

The overhead was reduced by more than 60% in all cases. In order to take the above measurements, we used a software instrumentation method. For EXT: Root to Simulator for example, we read the value of a hardware timer when an external event (e.g. new reflected light value) is detected.

Indeed, more messages are exchanged with E-CD++. If we examine more closely the first case - EXT : Root to Simulator - for example, E-CD++ will first add a X message and then a * message through the message admin that then processes them and send them to the flattened coordinator. This generates an X and * message to be sent to the simulator. Upon reception, the simulator calls the external method. In E-CDBoost, the runner adds the input message to the inbox of the flattened coordinator, calls the advance_execution() method, that leads to the simulator advance_execution() call that finally calls the external function of the concerned model. There are less generated messages in this case, and less storage/retrieval of messages involved. The future event list is more effective in E-CDBoost. For the output related events, we can observe that the overhead is less since less messages are involved (@ and Y) and no next event time computation is required.

Another set of tests, not related to this application and that would prove useful, is the case where multiple events are received in a short period. This is because CDBoost has proved to be very effective and achieved results comparable and sometimes better [16] than adevs, the fastest DEVS simulator according to a recent survey [19].

7. CONCLUSION

Using model-driven development for embedded systems is certainly a promising solution since the complexity and heterogeneity of the system are handled earlier in the development cycle. DEVS, in particular, with its formal nature and integrated time concept captures the essential characteristics of embedded systems.

We presented E-CDBoost used to build DEVS-based embedded applications. E-CDBoost is OS independent; it controls model execution on the target platform and interacts with the surrounding environment. It allows models to become controllers running on the execution platform. The internal structure of the RT executive separates the construction from the execution mechanism. It provides classes to the user in order to implement DEVS models easily. The execution mechanism, hidden from the user, renders the models behavior.

A case study was presented to provide a practical view of the development cycle and the usability of the new bare-metal kernel. The line tracking robot application was developped using E-CDBoost and the resulting binaries deployed on a Nucleo board mounted on the Seeed Studio Shield Bot. E-CDBoost allowed us to have a small footprint and reduce the message processing overhead by more than 60%.

REFERENCES

1. S. J. Mellor, T. Clark, and T. Futagami, “Model-driven development: guest editors' introduction.” *IEEE software*, vol. 20, no. 5, pp. 14-18, 2003.
2. B. P. Zeigler, H. Praehofer and T. G. Kim, Theory of modeling and simulation, Academic press, 2000
3. D. Niyonkuru and G. Wainer, "Discrete Event Methodology for Embedded Systems." *Computing in Science & Engineering* vol.17, no.5, pp.52-63, Sept-Oct. 2015.
4. M. Moallemi, G. Wainer, A. Awad, D. A. Tall “Application of RT-DEVS in military”. Proceedings of the 2010 Spring Simulation Multiconference, Orlando, FL, 2010.
5. A. Furfaro and L. Nigro, “A development methodology for embedded systems based on RT-DEVS” *Innovations in Systems and Software Engineering*, vol. 5, no. 2, pp. 117–127, 2009.
6. X. Hu and B. P. Zeigler, “Model continuity to support software development for distributed robotic systems: A team formation example” *Journal of Intelligent and Robotic System*s, vol. 39, no. 1, pp. 71–87, 2004.
7. A. Furfaro and L. Nigro, “Embedded control systems design based on RT-DEVS and temporal analysis using UPPAAL” International Multiconference on Computer Science and IT, Wisla, Poland, 2008.
8. G. A. Wainer, Discrete-event modeling and simulation: a practitioner’s approach. CRC Press, 2009.
9. A. C. Chow, “Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator” Transactions of the SCS, vol. 13, no. 2, 55–68, 1996.

10. Y. H. Yu and G. Wainer, “ecd++: an engine for executing DEVS models in embedded platforms” in Proceedings of SCSC, San Diego, CA, 2007.
11. A. C. Chow, "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator” *Transactions of the SCS*, vol. 13, no. 2, 55-68, 1996.
12. Y. K. Cho, X. Hu, and B. P. Zeigler, “The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems” *Simulation*, vol. 79, no. 4, pp. 197–210, 2003.
13. F. Bergero and E. Kofman, “PowerDEVS: a tool for hybrid system modeling and real-time simulation” *Simulation*, vol. 87, no. 1-2, pp. 113–132, 2011.
14. X. Hu, B. Zeigler, and J. Couretas, “Devs-on-a-chip: implementing DEVS in embedded java on a tiny internet interface for scalable factory automation” in Proceedings of the 2001 IEEE SMC, Tucson, AZ 3051–3056, 2001.
15. D. Niyonkuru and G. Wainer, "Towards a DEVS-based Operating System”. Proceedings of the 3rd ACM Conference on SIGSIM-PADS, London, UK, 101-112, 2015.
16. D. Vicino, D. Niyonkuru, G. Wainer, and O. Dalle, "Sequential PDEVS Architecture” Proc. TMS/DEVS 2015. Alexandria, VA, 2015.
17. Shield Bot V1.2. 2016. Retrieved February 26, 2016 from http://www.seeedstudio.com/wiki/Shield_Bot_V1.2
18. Advanced RealTime Simulation Laboratory. 2015. Line Tracking Robot on Embedded CDBoost. Video. Retrieved August 1,2015 from http://youtu.be/BZzzeJAa-cA
19. R. Franceschini, P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, D. Hill, R. Neykova and N. Ng "A survey of modelling and simulation software frameworks using Discrete Event System Specification” in *Imperial College Computing Student Workshop*, London, UK, 2014.