# Towards a DEVS-based Operating System

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

## ABSTRACT
Embedded systems are becoming increasingly complex and heterogeneous. Formal methods have proven effective in ensuring reliability and safety. However, they are hard to scale up. Modeling and Simulation (M&S)-based methods, on the other hand, deal effectively with scalability issues and provide the benefits of a risk-free testing environment. Yet, they are usually at most semi-formal, and models are not directly executed on the target hardware. To address the above challenges, we present a formal M&S-based kernel that runs on bare-metal and execute the original simulation models on the target hardware.

## Categories and Subject Descriptors
C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.13 [**Software Engineering**]: Reusable Software – Reuse models; D.4.7 [**Operating Systems**]: Organization and Design – *Real-time systems and embedded systems*; I.6.8 [**Simulation and Modeling**]: Types of Simulation – *Discrete event*; B.1.2 [**Control Structure and Microprogramming**]: Control Structure Performance Analysis and Design Aids – *Formal Models*; B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids – *Formal Methods*;

## General Terms
Design, Experimentation, Theory.

## Keywords
Real-Time Embedded Systems; Model Execution Engine; DEVS;

## 1. INTRODUCTION
Embedded systems are everywhere, and they shape the world. Any device that runs on electricity either already has, or will soon have a computing system embedded within it. An embedded system is generally defined as a combination of computer hardware and software, designed to perform a dedicated function. Real-Time Embedded Systems (RTES) in particular, in addition to producing correct responses, are also required to deliver them within strict timing constraints [11]. Missing these deadlines may lead to significant loss and in some cases catastrophic consequences. Other constraints related to these systems are limited dimensions, low cost, and low power requirements.


*SIGSIM-PADS'15*, June 10–12, 2015, London, United Kingdom.
 ISBN 978-1-4503-3583-6/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2769458.2769465

In addition to dealing with timeliness requirements, RTES design needs to deal with hardware/software partition, and cope with target systems' increasing scalability and complexity. However, there had been a real shortage of effective design and implementation practices. Most design methodologies are ad-hoc based, and therefore hard to scale up for larger systems, and/or require tremendous testing effort with no guarantee of a bug-free software product. Deficiencies come from two main weak areas: the development cycle and system verification. Indeed, disruptions exist in the development cycle since different artifacts and tools are used throughout the various phases [10]. System verification, on the other hand, is hardened by these discontinuities as well as the absence of robust development framework.

Recently, formal methods have shown great potential in dealing with these issues [17], but these methods remain hard to scale up. On the other hand, model-based design techniques handle well heterogeneity but the lack of formal modeling and effective model transformation are major roadblocks. A practical solution to the above problems is the use of formal Modeling and Simulation (M&S), therefore combining the advantages of a simulation based approach with the rigor of a formal methodology [16].

Such a M&S-driven approach must, however, ensure efficient model transformation, and should especially allow the original models to run on the target hardware. In this paper, we will present a kernel based on a formal M&S methodology that enables the user to run models directly on bare-metal. The objective is to be able to execute models directly on the target system hardware without the need of an operating system. The new model execution engine presented here provides functionalities similar to those of a real-time kernel, with formal models operating as system processes. This step narrows further the gap between the simulation and implementation phases. In fact, the same models are used for both simulation and execution on the final target. In order to show the feasibility of the approach, we present a case study of a line tracking robot using the bare metal environment.

A kernel that allows models to run on bare-metal was developed, and tested on ARM Cortex-M boards. As an application, we have modeled, simulated and deployed a line tracking robot. The results obtained using the new environment are compared and validated against another existing embedded environment.

## 2. BACKGROUND
The proposed approach is based on DEMES (Discrete-Event Modeling of Embedded Systems) [3] that offers a practical method in which models are consistently used throughout the development cycle. DEMES is an M&S based development methodology based on Discrete-EVent Systems specification (DEVS), which is a discrete event simulation formalism for modeling and simulating dynamic systems. The DEVS formalism [15] decomposes complex system designs into basic (behavioral)

models called atomic, and composite (structural) models called coupled. Precise rules are followed to define state changes of the modeled system with regards to input events or time delay triggers. DEVS is especially suitable for RTES since it provides a rich structural representation of components, and formal means for explicit time specification, which is essential to RTES. It has proven to be successful in different complex systems and can also be used alongside with existing real-time techniques such as state-charts, VHDL, Verilog and Timed Automata [17] [18].

## 2.1 DEMES

DEMES uses M&S for the initial stages, and replaces models incrementally with hardware surrogates without modifying the original models. The transition can be done in incremental steps, incorporating models in the target environment after thorough testing in the simulated platform, allowing model reuse throughout the process.

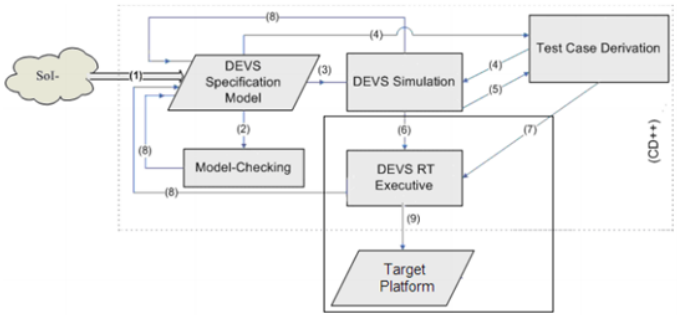


**Figure 1. DEMES Development Cycle [17]**

A DEMES based development cycle [17] [18] involves Model Specification, Model-Checking, DEVS Simulation, DEVS Real-Time Execution on a Target platform, and Testing. In the **model specification** stage, we define a specification model of the System of Interest (1) using a formal model (using DEVS or alternative techniques translated to equivalent DEVS models). After model specification, **model-checking** (2) can be used for model properties validation. The same models are then used to run **DEVS simulations** (3) of the behavior of the different sub models under specific loads. In brief, we first study system properties analytically, and complement the proofs using simulation, which can also be used for hardware/software co-design (and for training). The same DEVS specification model is used to derive **test cases** (4) (5), which can be also used for the simulation studies. Deriving test cases from both the model and from the simulation results allows us to check that the models conform to the requirements. Once we are satisfied with both analytical and simulated results, the models are incrementally moved into a target platform. A **real-time executive** (6) executes the models on the particular hardware (9). If the hardware is not readily available, the software components can still be developed incrementally and tested (7) against a model of the hardware to verify viability and take early design decisions. As the design process evolves, both software and hardware models can be refined, progressively setting checkpoints in real prototypes. At this point, those parts that are still unverified in the formal and simulated environments are tested, increasing the confidence of the engineer into the implemented system. Any modifications require going back to the same model specifications (8), which ensure that we can provide a consistent set throughout the development. This software lifecycle is cyclic, allowing refinement following a spiral approach. On each cycle of the spiral, we end with a prototype application consisting of software/hardware components interacting with simulated components.

Other M&S based frameworks and methodologies such as UML-RT, Ptolemy II, ECSL and Matlab/Simulink have been developed but they are semi-formal (which makes more difficult proving valuable properties about the models under development), and do not provide model continuity in the RTES development lifecycle [12]. Instead, formal modeling methods like DEVS provide sound syntax/semantics for structure, behavior, time representation and composition, which lend themselves to well-defined computation**.** Plus, the DEMES approach offers the following advantages [17]:

- Reliability: logical and timing correctness rely on DEVS system theoretical roots and sound mathematical theory.
- Model Reuse: DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition.
- Hybrid modeling and knowledge reuse: different methods can be used while keeping independence at the level of the executive, using the most adequate technique on each part of system architecture and reusing existing expertise.
- Process Flexibility: hybrid modeling capabilities are transparent for the executive, which is defined by an abstract mechanism that is independent from the model itself.
- Testing: several tests can be carried out and the definition of experimental frames can be automated.

## 2.2 DEVS-based frameworks

The DEMES concepts have been applied in the development of different tools to offer a unified and consistent development environment. Existing DEVS based development environments for RTES include DEVSJAVA [7], a Java DEVS-based simulator that supports high-level modeling; RTDEVS/CORBA [5], a DEVS implementation based on real time CORBA communication middleware; PowerDEVS [3] a tool for hybrid system modeling and real time simulation; and E-CD++[12][18],an engine for executing DEVS models in embedded systems. The platform limitations remain significant compared to the existing methods: In [7], [5], [9] and [4] where implementation requires Java, the target hardware should be able to support the Java-implemented DEVS real-time execution environment. In [8], the authors presented a DEVS based real-time system on a TINI chip which has limited memory and processing ability. However, this requires Java Virtual Memory and Java class libraries availability on the chip. In [3], Linux RTAI kernel is required for PowerDEVS. In [12], our implementation relied on Xenomai/Linux kernel services. Therefore, although the DEMES approach offers multiple benefits, tools have to be improved to overcome limitations and support different hardware.

The E-CD++ developed by our team [12] was used in various applications and relied on a variant of the Linux kernel. Xenomai provided hard real-time functionality to the Linux kernel. In this paper, we go further with bridging the gap between simulation and implementation (enabling the utilization of the same models) by removing OS limitations while decreasing the embedded application footprint, and increasing efficiency and portability. In the next section, E-CD++ software components will be presented and its implementation explained.

## 2.3 E-CD++

The DEVS formalism proposes a framework for model construction and defines an abstract simulation mechanism that is 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.

E-CD++ [18] is a real time implementation, based on the CD++ simulator [15] [14] (a DEVS-based framework for M&S), and RT-CD++ [16] (an extension of CD++ for real-time simulation). E-CD++ supports modeling real-time systems by converting the CD++ virtual time-advance function to real-time, and provides an RT simulation platform for verification of such models.

Figure 2 illustrates the E-CD++ development framework. The embedded platform with the external environment is shown in this layered approach representing the cross-platform development of models. The modeller defines models using a high-level DEVS language combined with C++ code if needed, which provides the application layer. These Real-Time models are then interpreted and executed by the DEVS Real-Time (DEVSRT) engine [12].

To allow for direct replacement of models with external entities, the I/O ports of E-CD++ models implement the formal interfacing mechanism of DEVSRT in the Driver Interface layer. The underlying middleware is a real-time kernel and the runtime objects are imported to this platform as RT tasks. In the previous iteration, the E-CD++ execution engine used the Xenomai real-time kernel [12] with multi-tasking services to implement DEVSRT. The user models and the driver objects were merged with the E-CD++ core objects; and the entire combination was compiled to produce an executable.

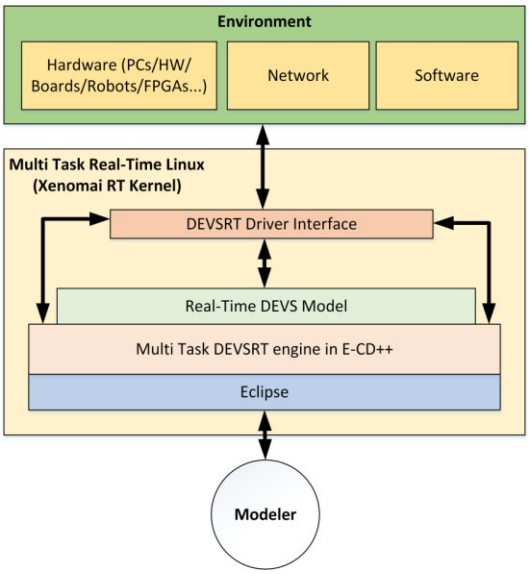


**Figure 2. E-CD++ Layers [12]**

E-CD++ also include several features also. The Eclipse IDE layer shown in figure 1 also allows for the graphical development of models. Through the IDE, the Generic Graphical Advanced environment for DEVS modeling and simulation (GGAD) [18] allows the developer to use a graph-based representation to specify models hierarchy, interconnections and behaviors to automate model generation. At the execution engine level, various features have been implemented in order to improve the software including DEVSRT simulation algorithms, a Flattened Coordinator technique and a Time Interval function. The P-DEVS simulation algorithms allows for parallel execution of concurrent events through the implementation of a messaging behaviour for model interaction. The Flattened Coordinator technique improves the efficiency of the DEVSRT messaging behaviour through the removal of superfluous messages that are generated for communication between coupled models. Finally, the Time Interval function enforces real-time constraints through the use of wall-clock time advancement and execution deadline checking.

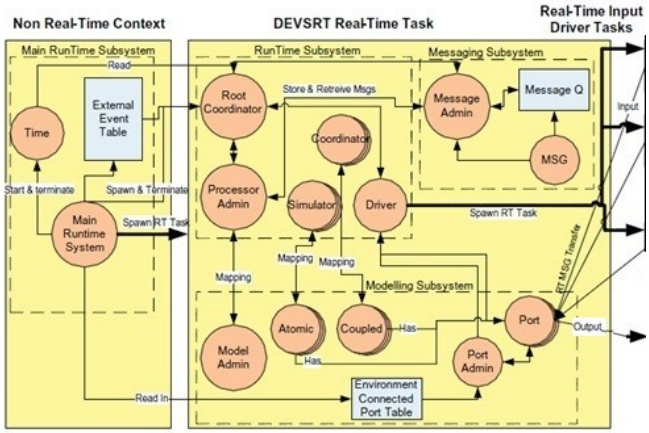


**Figure 3. Software Components [12]**

CD++ has four main components [Figure 3]: the Main Runtime System, the Modeling Subsystem, the Runtime Subsystem and the Messaging Subsystem [17]. The Main Runtime System manages the overall aspects of the real-time execution and provides timing functions with microsecond precision. The Main Runtime System is the first object that is created in non-real-time context, and it launches the Runtime Subsystem [12]. In general, the Main Runtime System first register Atomic component objects, then the Top coupled component ports that are connected to the external environment, reads in the external events (from an existing event-file) and builds an external event table. After that, the Main Runtime System reads in the model-file and builds the model hierarchy. Finally, it spawns the main real-time task in which the Root Coordinator (RC) is created to start the DEVSRT execution cycle. The Runtime Subsystem consists of Simulators, Coordinators, and the Processor Admin. In E-CD++, the Simulators work on run-time engines that correspond to atomic components, and they perform the main job of executing the internal transition and output function after receiving the proper messages.

The RC is a special Coordinator that manages the real-time event scheduling. It initializes the global Driver object which launches the real-time input driver tasks (associated with input ports of the Top coupled component in the DEVS model hierarchy) declared by the user.

The Modeling subsystem is generated in order to define the atomic and coupled models, as well as the relationships between them. For each of these models, a processor is defined within the Runtime Subsystem in order to manage the behavior of the model and drive the execution. The Messaging subsystem provides the P-DEVS behavior [12].

While this implementation reflects DEMES concepts, it is closely dependent on the Linux kernel and restricts supported devices. We went further and removed this limitation: The new E-CD++

provides a DEVS execution engine that resides in an ARM-based microcontroller, and it is OS independent. Today, the ARM architecture is the most pervasive 32-bit architecture and is found in all types of computing devices from real-time safety systems (automotive braking systems) to smartphones [13]. Besides, since the new E-CD++ does not rely on a particular OS, it becomes applicable to a broader variety of microcontrollers. In order to successfully achieve this objective, several changes were required and will be presented in the next section.

## 3. A DEVS-based Kernel

As discussed in Section 2, the development of embedded applications using E-CD++ requires several changes to the current iteration of the software. Currently, the E-CD++ execution engine relies on a RT-Linux OS to implement DEVSRT [12]. Through modifications to the existing software architecture, we provide now stand-alone operation, i.e. bare-metal execution. To do so, we had to leverage multiple existing functions as well as develop additional functionality in order to operate without OS support and directly interface with hardware devices.

### 3.1 Proposed Approach

As described earlier, the current implementation of E-CD++ is based on the assumption that it will be running from a variant of the Linux kernel. This imposes memory capacity, processing, and portability limitations as the target platform must include the memory and processing power necessary for the Linux kernel variant, and there must be a Linux kernel that can be compiled for the target platform and can interface with the available hardware devices.

As a solution to the above challenges, we propose the new architecture shown in figure 4. The modeller defines models using the DEVS formalism and C++ code. Note that an Eclipse IDE can be used in order to make the development task easier. The defined models are then interpreted and executed by the DEVSRT engine which directly rest on bare-metal (target platform) in the new version. Similarly to the previous version, the driver interface layer is used to provide a formal interface for I/O ports. However, this layer has been modified to communicate directly with the underlying hardware without the need of a real-time kernel middleware.

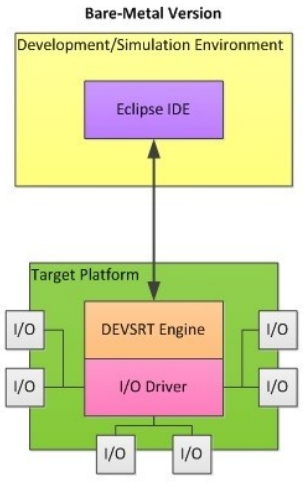


**Figure 4. New Layers**

Besides, the simulation platform is only used for model development since the DEVSRT execution engine and the I/O drivers are moved to the target platform and run on bare-metal. This is new when compared to the previous platforms where a RTOS was required, and it is especially different from the case where commands were sent through a network interface as shown in figure 2 and in figure 4. With these changes, models execute in real-time on the target platform and model continuity is greatly increased providing therefore higher integrity simulation results.

To develop this DEMES-based design solution that is able to execute directly on the target platform, several updates were made including the reimplementation of a Driver model, and the use of the Flattened Coordinator technique. Plus, the previous implementation of E-CD++ used the Xenomai real-time framework for the Linux kernel; this dependency was removed by interfacing with the hardware clock of the target platform.

E-CD++ has to be implemented as a stand-alone embedded program able to cope with memory limitations of embedded platforms and run independently on the generally limited RAM or ROM available for program storage and execution. Because of this resource limitation, it is necessary to reduce the memory footprint of E-CD++ as much as possible.

More specifically, to implement this novel approach, we have first adapted the Driver model and the Flattened Coordinator technique for direct I/O interfacing as opposed to the transfer of data over a network. The Driver model and Flattened Coordinator concepts have been leveraged in the new E-CD++. Because embedded devices generally lack high memory limits and processing power, the Flattened Coordinator was used to minimize the amount of message passing between models. The driver model, on the other hand, provides a programming construct to be used with hardware components and devices and was adapted to the new architecture.

Second, to migrate E-CD++ to a bare-metal environment, several changes were made. These changes include the removal of the RTOS as well as the reimplementation of various OS kernel function calls. Since E-CD++ will be implemented directly on an embedded platform, the additional functionality provided by the RTOS becomes redundant due to the available low level control of timing and scheduling and as there is only one application running at a time. With these changes, it is possible to move the DEVSRT engine as well as the I/O drivers directly onto the target platform, eliminating the need for a network interface for communication between the simulation platform and the target platform as illustrated in figure 4. Hence, the removal of the Xenomai/Linux RTOS minimizes the size of the program and removes previous Linux dependencies.

Last, as the initial E-CD++ was developed for targets with OS support, there were various system calls made to the Linux kernel in order to handle various functions such as file I/O and memory management. In order to minimize the footprint of E-CD++ to allow for execution in a low memory embedded platform, these functions need to be re-developed with embedded execution in mind. As is discussed later, these calls are largely unnecessary as the embedded platform will not support multi-processing and does not require full file system support; the only file referenced by E-CD++ is the model file which can be loaded directly into memory. The new functions were thus implemented to provide the same functionalities as the original system calls without the overhead of a full OS kernel.

In the following section, the above main changes will be detailed and their implementation further explained.

## 3.2 Implementation

In a first phase, two main concepts, namely the driver concept and Flattened Coordinator Technique, were adjusted to the new structure. The Run-Time Subsystem and Modelling Subsystem now include a Flattened Coordinator as well as the integration with the Driver model. The Flattened Coordinator has been added to the Run-Time Subsystem where it has replaced the many Coordinators that were previously used. The Flattened Coordinator reduces the number of messages that need to be passed between models. This is accomplished through the removal of Coordinators for Coupled Models which are replaced with a single Flattened Coordinator, which manages the message passing between Atomic models enabling direct communication, reducing the messages needed. This improves processing power and speed, which is a limitation to execution on embedded platforms. In order to increase efficiency, the Flattened Coordinator analyzes the links between models on initialization and generates an influence list, establishing the relationships between models. The Flattened Coordinator is able to identify the recipient of a message and passes the message directly to the Atomic model. In a simple system containing only a few coupled models, this will not have a very large effect on the overall efficiency of the system; however, as the system complexity increases, the increase in performance that is achieved through the implementation of the Flattened Coordinator technique can be seen to improve [18].

Drivers have also been added to the Run-Time Subsystem providing an interface to initialize hardware devices as well as to interact with the Ports that are associated with the model as it will be discussed later. Through the use of the Driver objects, external I/O can be controlled through the encapsulation of hardware specific functionality, made available by accessing the generic functions provided by the Driver model, and not using an embedded RTOS. Using this model, we are able to interface a wide range of devices and greatly improve portability by simply implementing basic hardware drivers that are then accessed by E-CD++. From an implementation point of view, the driver model manages hardware device connections and interfacing through the use of two classes: the Port class, and the Driver class.

Similar to the Models and Processors in the Modelling subsystem and Run-time subsystem (seen in section 2.3.), the Port class resides in the Modelling subsystem while the Driver class is in the Run-time subsystem. Together, they provide a link between the DEVS implementation, and the hardware target platform.

The Port class represents the logical connection between models and hardware devices. Where the previous implementation saw the Port class passing established API commands over a network interface, our implementation of the Port class includes Input and Output low-level functions that are developed to provide the interface directly with the hardware device. In the case of Inputs, the receipt of a signal on a Port will cause the generation of a PDEVS message which is then added to the message queue processed by the Root Coordinator. When configured as an output, the Port will receive the data from a PDEVS message from the Driver class which will then be translated into a signal that can be interpreted by the hardware device. Through bare-metal implementation, it is possible to use the hardware and software interrupt service routines of the target platform to notify E-CD++ of I/O changes. The specific hardware interrupts associated with each hardware device can then be used to generate PDEVS messages based on the values received from the device. Similarly, software interrupts can be programmed based on a division of the base clock in order to provide periodic polling.

Alternatively, the role of the Driver class is to receive PDEVS messages from the message queue as well as initialize and close hardware devices. As mentioned, when a PDEVS message is retrieved from the message queue, the Driver reads the value from that message and passes it on to its associated Port for interpretation and communication to the device. In the case of initialization or termination, the Driver class includes functions that interface with hardware devices in order to prepare them for operation, or for the end of simulation as required.

In addition to the above, to effectively model real-world inputs, it is necessary to define two types of devices that a Port/Driver may be associated with, the first being *passive* devices. These types of devices include sensors which must be polled at specific intervals to determine their current state. Interfacing with passive devices requires the implementation of a periodic timer interrupt that requests the state of the device. This can be accomplished through the creation of a software interrupt that is tied to a division of the base clock. This allows a software interrupt to be triggered at regular intervals, eliminating the need for real-time tasks. The state that is returned from these interrupts is then passed to the associated Driver which interprets the state, creates, and sends an appropriate PDEVS message to the message bag for further processing. The second type of hardware device that can be seen is an *active* device. An active device is classified as a hardware device that triggers an input event. Active devices can trigger a hardware interrupt at which point, they will pass their states to the Driver for processing.

All operating system dependencies needed to be removed. The original implementation of E-CD++ was developed on a desktop computer for simulation, which would then pass the results of the simulation through the OS to a separate application either on the same platform or over a network to the target platform. With the new version, the DEVSRT execution engine resides directly on the target hardware. As embedded platforms are generally limited to several megabytes of memory and a processor with a clock speed in the megahertz, the OS dependency of the C++ library was removed in order to streamline the performance of the software as well as increase the portability of the overall system. Given that E-CD++ will be the only application that is running on the target platform, there is no need for the extra capabilities available from OS inclusion.

We were able to quickly determine the system calls that were being made. Based on this list, it was possible to identify the purpose and functionality of each of these calls. The functionality of each of these functions could then be reproduced through the creation of functions with the same signature but with a re-designed implementation that takes into accounts the limitations and environment of the target platform. Among these functions, several were deemed unnecessary as they pertained to inter-process communication within a multi-processing system. While they were still required for the compilation of the E-CD++ executable, these functions were re-developed to return constant, known values that are similar to what would be expected when running from within an OS. An example of one of these functions

is the *getpid* function. The purpose of this function is to return the process ID of the currently running process. As there is only a single process running, the value returned by this function can be set to an arbitrary integer that meets the constraints of what would be expected from an application launched from an OS. Multi-processing and multi-programming can be implemented directly as a DEVS coupled model, which can be formally verified using model-checking, and building individual kernels for particular purposes.

Although the functions related to inter-process communication could be easily removed, there were still several functions that required significant re-development in order to return appropriate values given the context in which they would be called. Some of these functions relate directly to the programming language that is used, others to the functionality that is provided by the OS. For instance, as E-CD++ is developed in C++, an object-oriented language, dynamic memory allocation is required in order to allow for the instantiation of new objects. While this can be accomplished through the run-time modification of pointers to heap memory allocation, other OS specific functionalities are not so easy to replace, becoming a hurdle to the bare-metal implementation of E-CD++.

We also needed to provide useful functionalities generally provided by the OS. This involved the implementation of several functions that take advantage of the hardware available to replicate the OS functionality. Through the use of hardware devices available on the target platform, such as a real-time clock, on-board memory, and low power modes, the replication of key OS functionalities and complete removal of the OS becomes possible.

In the previous version, Xenomai provided real-time guarantees through the implementation of constrained functions as well as a real-time scheduler. More specifically, while the previous version of E-CD++ used the Xenomai real-time framework for Linux to provide real-time constraints and scheduling, E-CD++ will not have these capabilities available. Instead, timing can now be controlled at the clock level through the creation of periodic software timer interrupts in order to manage scheduling, and at the model level through model specification and model-checking of the timing constraints. As E-CD++ requires microsecond precision, a software timer can be defined that is set to trigger at 1 MHz. By causing this interrupt to commence the next simulation cycle, simulation can occur as it normally would. With this, it is important to note the introduction of a minimum processing speed requirement for the proper execution of E-CD++. Because the simulation cycles are defined as 1 μsec, the clock speed of the microcontroller must be greater than 1 MHz in order to allow the execution of each simulation cycle prior to the next timer interrupt.

Regarding hardware I/O, the implementation of drivers provides hardware I/O interfacing by implementing basic hardware drivers that can be easily accessed by E-CD++. As previously mentioned, this is accomplished through the use of interrupts and hardware polling. Hardware and software interrupts can be used to generate messages from active devices; when the interrupt is triggered, a message can be added to the message bag. One complication that arises in this case is the object-oriented support that is available in C++ but not in C. This is further complicated by the name mangling that occurs with C++ functions. For this reason, it was necessary to generate a wrapper function written in C++ but with a C signature. This function can then call the C++ functions necessary to add the message to the message bag. Passive devices are simpler in that it is only necessary to develop the initial interface functions in C. These functions can be called from C++ whenever input is necessary without any problems.

The only file referenced by E-CD++ during execution (and thus needed on the target platform), is the model file. In fact, models are loaded into E-CD++ at run-time through the reading and interpretation of a model file. This is done by providing E-CD++ with the name and location of the model file from the command line. Since we do not have a directory structure for OS file I/O support, it was necessary to develop a pseudo file system in order to maintain continuity between desktop simulation and target simulation. In order to mimic this behaviour, the model files are loaded directly into memory and the file names are used to populate a file register. The file register then determines the memory address of the text file using a file table which contains the mapping between file names and memory addresses. The file table also provides information about the file that is required by the C++ library, for example, the file size.

In addition, the removal of the Xenomai framework also required redevelopment of the behaviour of the Root Coordinator's handling of done messages. With Xenomai, the Root Coordinator would wait to receive a message from a Xenomai real-time task indicating hardware input. Since this functionality is no longer available, the receipt of the done message will cause the Root Coordinator to sleep until the next internal transition is scheduled, periodically verifying that an external event has not occurred. If an external event occurs, the event will be processed prior to the internal transition and the cycle will repeat. In the case where there are no more internal transitions scheduled, the Root Coordinator will place the microcontroller into a low power mode and await an external event.

Finally, early integration of stand-alone E-CD++ was done using an MCBSTM32F200 evaluation board. Developed by Keil, the board includes the STM32F207IG ARM Cortex-M3 based microcontroller. This microcontroller has a clock speed of 120 MHz and contains 1 MB of ROM and 128 KB of RAM. The clock speed meets the 1 MHz requirement and the memory capacity is great enough to hold the E-CD++ application and associated model files. Through the implementation of drivers for the LEDs and buttons contained on the evaluation board, early integration testing was performed and proved to be successful, demonstrating the feasibility of bare-metal implementation of E-CD++. On the software platform side, Eclipse was used along with the GNU ARM bare metal tool chain to build applications and GDB to debug hardware and software. .

Overall, a high level of portability and model continuity can be achieved, as the DEVS model is not changed throughout development. This design is also portable as the software core of E-CD++ has not changed; all that has changed is the external interfaces. As mentioned, the implementation of the Driver model greatly increases this portability through the encapsulation and generalization of I/O devices allowing for simple addition of new devices.

In the upcoming section, we will illustrate how the new E-CD++ version can be used by implementing a line tracking robot behavior and describing the entire software development process using our DEMES-based approach

# 4. Case Study: A Line Tracking Robot

In this section we will show an example of the use of the new bare-metal version of E-CD++, by building a case study application for a line tracking robot controller. The Robot is equipped with a light sensor that faces the ground and absorbs the amount of 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 it does not pick up a path trail, it stops, turns counter-clockwise slightly, and then tries to detect a trail again. If a path is detected, the robot will move forward again; otherwise it will continue to turn until it finds a path to follow. The destination is considered to be a wide dark ground surface. At that point, the light sensor would detect a small amount or no light reflection which indicates to the robot's controller that it has reached the destination and causes the robot to stop moving. The robot can also receive manual signals to start and stop.

## 4.1 System Architecture

The first step in the DEMES-based development cycle is to specify a model of the system of interest using DEVS. Figure 6 illustrates the resulting DEVS model hierarchy for this example.

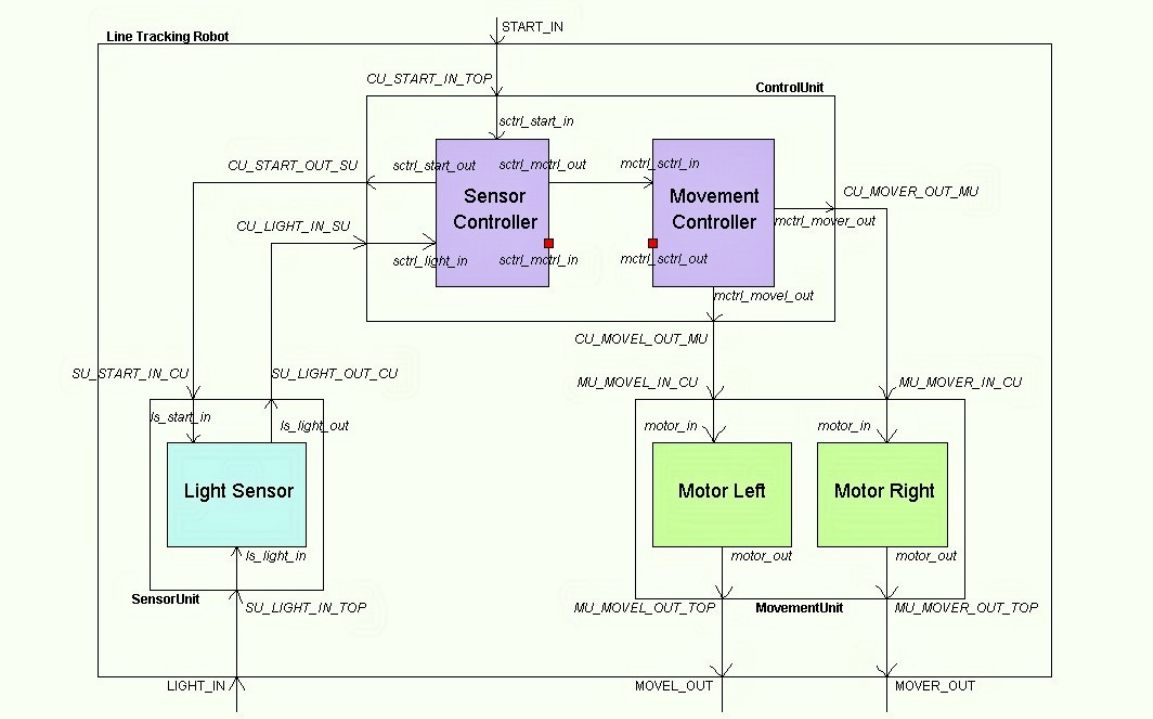


**Figure 6. Line Tracking Model Robot Hierarchy Diagram**

The top model has one input port, LIGHT_IN, through which the light sensor values are read, and two output ports, MOVEL_OUT and MOVER_OUT, used to send commands to the left and right motors of the robot. Apart from these ports, other input ports are named using the format *ToPort_IN_FromModel* where *ToPort* represents the name of the input port to which a message is sent and *FromModel* is the coupled model from which the message originates. Note that when the signal comes from an atomic model, the *FromModel* part is omitted, and the format becomes therefore *InputPortName_IN*. Hence, *motor_in* is the input port of the motor atomic model. Similarly, for output ports, we use the *FromPort_OUT_ToModel* where *FromPort* is the port through which a message is sent, and *ToModel* is the coupled model the message is adressed to. When the recipient model is atomic, the format is reduced to *FromPort_OUT*. For instance, MU_MOVEL_OUT_TOP is the Movement Unit(MU) output port designed to send messages from the left motor to the TOP model. On the other hand, *mctrl_mover_out* is the mover (move right) Movement Controller(mctrl)'s output port. These formats were used in order to rapidly identify the links among models and the role of each port.

In terms of components, the Line Tracking Robot's top model is a coupled model made of three coupled models: *Sensor Unit*, *Control Unit* and *Movement Unit*. The *Sensor Unit* contains an atomic model: *Light Sensor*. The Light Sensor reads the amount of light reflected off of a ground surface and transmits the readings to the *Control Unit* for processing. The *Control Unit* contains two atomic models: *Sensor Controller* and *Movement Controller*. The *Sensor Controller* activates/stops the light sensor through the *sctrl_start_out* port, receives the light sensor readings through *sctrl_light_in* and sends messages to the movement controller through the *sctr_mctrl_out* output port. These messages specify whether the robot is on-track, off-track or has reached the destination. Indeed, when the light sensor readings indicate a bright surface, the sensor controller sends an off-track signal to the movement controller. Likewise, when the sensor readings indicate a dark surface, it implies that the line tracking robot is properly following the line and an on-track signal is sent instead. When the robot reaches its destination, i.e. the light sensor reads an all dark surface; the sensor controller sends a stop reading command to the light sensor through *sctrl_start_out* and a stop signal to the movement controller. In addition to the above, the

sensor controller receives, through the *sctr_start_in* input port, the user signal that starts the line tracking robot and puts the system in motion.

The Movement Controller, on the other hand, receives on/off-track signals from the sensor controller through the *mctrl_sctrl_in* port, and sends appropriate commands to the motors through *mctrl_movel_out* and *mctrl_mover_out*. Therefore, when an on-track signal is received, the movement controller sends a go forward command to both motors in order for the robot to stay on the right path. Correspondingly, when an off-track signal is received, the robot stops and prepares to turn. In this case, a stop signal is sent to all motors; then the right motor is instructed to go forward while the left motor is commanded to go into reverse.

Finally, the *Movement Unit* is made of two atomic models: *Motor Left* and *Motor Right*. 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 functions of the robot treads. They can only move forward, in reverse or stop according to the signals they receive from the *Control Unit*. A combination of a motor moving forward and the other motor moving in reverse makes the robot turn.

## 4.2 DEVS Model Specification

Once the hierarchical structures of the model have been established, components are well defined using DEVS formal specification. In this section, we will focus on the Control Unit model specification. As mentioned earlier, this latter is a coupled model that has two atomic models, the sensor and movement controllers.

The Control Unit can be formally defined as:

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

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 the above specification, X represents the set of input events (N being the set of port values), Y the set of output events, D the component name of each model, Md the DEVS basic (atomic or couple) model, EIC the external input coupling, EOC the external output coupling, IC the internal couplings and finally select is the tiebreaker function (refer to Appendix A for more details about this specification).

The DEVS formal specification of the *Sensor Controller* model is as follows and shows how atomic models are defined:

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

Where
**X**:{(sctrl_light_in,{BRIGHT,DARK,ALL_DARK});(sctrl_start_in,{START_PROC,STOP_PROC});(sctrl_mctrl_in, {Ø})}

**S**: {"IDLE", "PREP_RX", "WAIT_DATA", "TX_DATA", "PREP_STOP"}

**Y**: {(sctrl_mctrl_out, {ON_TRACK, OFF_TRACK, STOP_PROC}); (sctrl_start_out, {START_PROC, STOP_PROC})}

**$\delta_{int}$** (s) {

```
switch (s){
        case PREP_STOP:     // Stop request
                    state = IDLE; ta(state)=infinity;
         case PREP_RX:      //Preparing to read data
         case TX_DATA:    // Sensor transmitting data
                state = WAIT_DATA; ta(state)=infinity;
}
```

**$\delta_{ext}$**(s,e,x){

```
if (x.port() == sctrl_start_in){  // A user command is received
        if(state == IDLE && x.value()== START_PROC){
              state = PREP_RX; ta(state)= scRxPrepTime;
        }
        else if (x.value()== STOP_PROC) {
              state = PREP_STOP;  ta(state)= ZERO_TIME;
         }
}
else if (x.port() == sctrl_light_in){ // Reading from sensor
        if(state == WAIT_DATA) { //  Waiting for sensor data
             sensor_input = x.value();
             if(sensor_input == ALL_DARK) {// Destination
                  state = PREP_STOP; ta(state)= ZERO_TIME;
             }else {
                  state = TX_DATA;  ta(state)= scTxTime;
               }
}
}
```

**λ**(s) {

```
switch (s){
        case PREP_STOP:
              sendOutput( time, sctrl_start_out, STOP_PROC) ;
              sendOutput( time, sctrl_mctrl_out, STOP_PROC) ;
        case PREP_RX:
              sendOutput( time, sctrl_start_out, START_PROC)
        case TX_DATA: {
              int output_val;
              if(sensor_input == DARK)
                output_val = ON_TRACK;
              else if(sensor_input == BRIGHT)
                output_val = OFF_TRACK;
              sendOutput( time,sctrl_mctrl_out, output_val) ;
}
}
```

**ta**: $S \rightarrow R^{+}_{0,\infty}$ has been defined in the pseudocode above.

To understand the behavior of the *Sensor Controller* model, the following figure illustrates its state transitions using a state diagram: Fig. 7 illustrates the DEVS Graph representing the sensor controller's behavior. The state diagram summarizes the behavior of a DEVS atomic component by presenting the states, transitions, inputs, outputs and state durations graphically [36]. The circles represent states and the double circle is the initial state. The name and duration of a state is shown in the circle. The continuous edges between the states represent the external transitions, which includes 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").

The Sensor Controller starts in the IDLE state and remains in that state until a start command is received. If the user start signal is received, an external transition is triggered and the Sensor Controller state changes to PREP_RX. At this stage, it waits for a defined time ta=scRxPrepTime after which a 'start' output signal is sent to the Light Sensor and an internal transition is triggered changing its state to WAIT_DATA. The Sensor Controller waits in this state until it receives a signal from the Light Sensor. When a signal is received, if the signal indicates that the robot reached the destination (signal value is ALL_DARK), an external transition causes the Sensor Controller to go the PREP_STOP state, at which it will immediately send a stop signal to the Light Sensor and the Movement Controller and then transition back to the IDLE state. However, if the received signal is different, the Sensor Controller will go to the TX_DATA state at which it will wait for a time advance period of ta=scTxTime before it sends an output signal to the Movement Controller indicating whether the robot is on track or not, and transitions back to the WAIT_DATA state. At any point in time, if the Sensor Controller receives a manual stop signal (STOP_PROC), it will execute an external transition to the PREP_STOP state to stop all activities.

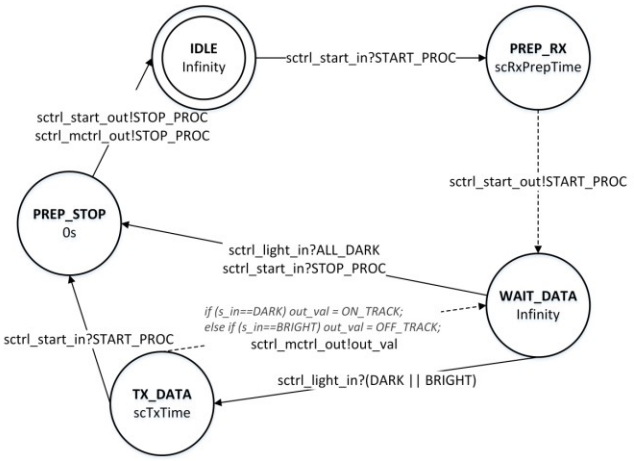


**Figure 7. Sensor Controller State Diagram**

After the formal specification phase, we implement the models using E-CD++ in order to run simulations, test individual components under different loads, gather results and derive different test cases.

## 4.3 Implementation in E-CD++

E-CD++ provides a mechanism to program DEVS models' hierarchical structures. The model definitions and couplings are written in a model file following a specific format, and the state transitions and output function are overwritten in C++, as part of each model's class definition.

The following lines describe the *Sensor* and *Movement Controllers* specification as components of the *Control Unit* in the model file (also called the MA file), in accordance with the model diagram in figure 6:

```
1  [ControlUnit]
2  components : SCtrl@SensorController
   MCtrl@MovementController
3
4  in : CU_START_IN_TOP CU_LIGHT_IN_SU
5  out : CU_START_OUT_SU CU_MOVER_OUT_MU
   CU_MOVEL_OUT_MU
6
7  %input connections
8  Link : CU_START_IN_TOP
   sctrl_start_in@SCtrl
9  Link : CU_LIGHT_IN_SU
   sctrl_light_in@SCtrl
10
11 %output connections
12 Link : sctrl_start_out@SCtrl
   CU_START_OUT_SU
13 Link : mctrl_moveR_out@MCtrl
   CU_MOVER_OUT_MU
14 Link : mctrl_moveL_out@MCtrl
   CU_MOVEL_OUT_MU
15
16 %internal connections
17 Link : sctrl_mctrl_out@SCtrl
   mctrl_sctrl_in@MCtrl
```

The MA snippet starts by defining the Control Unit as a coupled model composed of two instances: *SCtrl* and *MCtrl*, of *Sensor* and *Movement Controller* respectively. Then, the input (CU_START_IN_TOP and CU_LIGHT_IN_SU) and output(CU_MOVEL_OUT_MU and CU_MOVER_OUT_MU) ports of the *Control Unit* are defined. Finally, the input and output connections between the ports of the *Control Unit* and those of *SCtrl* and *MCtrl* are described, as well as the internal connections between *SCtrl* and *MCtrl*. The direction of the connection is read as FROM port → TO port.

The following code describes the transition and output functions of the *Sensor Controller*, in accordance with the state diagram in figure 7:

```
1  Model
   &SensorController::externalFunction(
   const ExternalMessage &msg ) {
2  …
3  if (msg.port() == sctrl_light_in){
   // Light sensor signal received
4     if(state == WAIT_DATA) {
   // Sensor controller was waiting for data
5      sensor_input= msg.value();
   // Get the light sensor input
6      if(sensor_input==ALL_DARK) {
   // Destination Reached
7        state=PREP_STOP;
   // Prepare to stop immediately
8        holdIn(Atomic::active,ZERO_TIME );
9      } else {
   // Robot is not at destination yet
```

```
10        state = TX_DATA;
    // Sensor goes into transmitting state
11        holdIn(Atomic::active, scTxTime );
    // after scTxTime,send data to MCtrl
12      }
13     }
14   }
15     return *this;
16  }
17
18  Model
    &SensorController::internalFunction(
    const InternalMessage & ) {
19     switch (state){
20            …
21            case TX_DATA:
    // Just transmitted data to movement
    controller
22                  state = WAIT_DATA;
    // Wait for new data from the sensor
23                  passivate();
    // stay in this state until new event
24                  break;
25     }
26     return *this;
27  }
28
29  Model &SensorController::outputFunction(
    const InternalMessage &msg ) {
30     switch (state){
31       …
32       case TX_DATA: {
    // in transmitting state
33         int output_val;
34
35         if(sensor_input==DARK)
    // light sensor indicates a dark line
36           output_val = ON_TRACK;
    // output signal to MCtrl is ON_TRACK
37         else if(sensor_input==BRIGHT)
    // light sensor reads a bright surface
38           output_val = OFF_TRACK;
    // send off_track signal to MCtrl
39
    sendOutput(msg.time(),sctrl_mctrl_out,
    output_val); // Output sent to MCtrl
40         break;
41        }
42     };
43     return *this ;
44  }
```

The code snippet first shows a portion of the external transition function that describes the transition from state WAIT_DATA to either TX_DATA or PREP_STOP depending on the value (sensor_input) of the incoming signal from the *Light Sensor* received on port sctrl_light_in. Lines 18 to 28 show a portion of the internal transition function describing the transition from TX_DATA to WAIT_DATA. Finally, lines 29 to 43 show a portion of the output function's behaviour at state TX_DATA. The output function sets the output signal (ON_TRACK or OFF_TRACK) to send to the *Movement Controller* through port sctrl_mctrl_out.

Using these classes and the core components of E-CD++, different scenarios can be tested early on the development platform namely by using event files that generates event for the input ports. Once the developer is satisfied with the results, the components can be incrementally moved to the target platform. In order to do this, each driver is associated with specific commands related to the hardware component it interacts with.

For the previous version of E-CD++, the Lego's NXT++ library was used to interface the models with hardware i.e. the light sensor and motors. Through a C++ API for Lego NXT robot controller, communication can be established over USB and Bluetooth. Therefore, hardware events can be monitored and events mapped to external transition functions. USB communication was done using Xenomai 2.6 and NXT++ v4.0 since v0.6 does not support Linux. In the same way, the E-CD++ to NXT++ interface can be used for translating hardware commands by having output functions mapped to NXT++ API. However, the NXT robot must be tethered and the DEVS models weren't compiled to the native NXT byte code.

The same models were deployed on the ARM board. This time, the native code was directly downloaded in memory via ST-LINK, an in-circuit programmer for the STM32 microcontroller families. This interface module is enabled with JTAG/serial wire debugging (SWD) interfaces that can be used to communicate with the target platform and debug via an OpenOCD client/server connection. Interfacing E-CD++ with hardware peripherals is made easy by the available port/driver concept and the comprehensive standard peripheral libraries offered by STMicroelectronics in this case. These two elements can be seamlessly integrated, compiled to the native byte code, and result in a DEVS-based firmware able to control the peripherals and respond to diverse external stimuli.

Once the models were implemented, different tests were done by progressively integrating hardware components and testing the entire system. The final deployment was made on the final target and models run on the microcontroller. Section 5 presents the results of our experiments and compares them with the Lego's version.

# 5. RESULTS

Each model component was first tested using virtual-time simulation. Then, multiple scenarios were simulated in order to observe the behavior of the robot in different environment settings and in real-time. To carry out these experiments, an event file that specifies the event time, input port and its value is used. Table 1 shows the port mapping table and the description of each value.

**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<br>1<br>2 | STOP<br>FORWARD<br>REVERSE | Stops the motor<br>Spins Clockwise<br>Spins Anticlockwise |

An example of events that were injected into the system follows:

```
00:00:01:000  START_IN  10
00:00:02:000  LIGHT_IN  1
00:00:02:500  LIGHT_IN  0
00:00:02:700  START_IN  11
00:00:03:000  LIGHT_IN  1
00:00:03:500  LIGHT_IN  0
00:00:05:000  START_IN  10
00:00:05:500  LIGHT_IN  0
00:00:06:000  LIGHT_IN  0
00:00:06:500  LIGHT_IN  1
00:00:07:000  LIGHT_IN  1
00:00:07:500  LIGHT_IN  1
00:00:08:000  LIGHT_IN  2
00:00:08:500  LIGHT_IN  1
00:00:09:000  LIGHT_IN  1
00:00:09:300  START_IN  11
```

After 1s, the system is started by sending an input to the START_IN input port. Then, at 2s, 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. Afterwards, the system is started again, and bright (0), dark (1) and all dark (2) surfaces are alternately sensed through the LIGHT_IN port. ALL_DARK signals that the robot has reached its destination and acts as an automatic stop signal. More values are sent through the LIGHT_IN port and finally a manual stop signal is sent through the START_IN port.

The resulting behavior is similar to the one defined in the controller models. Indeed, when the robot goes off track and does not detect the line, it stops, turns counter-clockwise slightly, and then tries to detect a trail again. If the line is detected, the robot will move forward again; otherwise it will continue 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.

The same inputs were used on the ARM board. Simulation results are below. Inputs are shown as well as their corresponding results (in bold). The format used is <time> <port> <signal_value>. Microseconds are shown in the logs since we used a 32 bit timer that allows such precision. Inputs are numbered and hardware commands instead of signal value. Output are shown in bold.

**1.** 00:00:01:000:023 START_IN START
**2.** 00:00:02:000:030 LIGHT_IN DARK
**00:00:02:200:119 mover_out 1**
**00:00:02:200:119 movel_out 1**
**3.** 00:00:02:500:021 LIGHT_IN BRIGHT
**00:00:02:600:115 mover_out 0**
**00:00:02:600:115 movel_out 0**
**00:00:02:700:115 mover_out 1**
**00:00:02:700:115 movel_out 2**
**4.** 00:00:02:700:027 START_IN STOP
**00:00:02:700:124 mover_out 0**
**00:00:02:700:124 movel_out 0**
**5.** 00:00:03:000:019 LIGHT_IN DARK
**6.** 00:00:03:500:030 LIGHT_IN BRIGHT
**7.** 00:00:05:000:027 START_IN START
**8.** 00:00:05:500:021 LIGHT_IN BRIGHT
**00:00:05:700:115 mover_out 1**
**00:00:05:700:115 movel_out 2**
**9.** 00:00:06:000:028 LIGHT_IN BRIGHT
**10.** 00:00:06:500:022 LIGHT_IN DARK
**00:00:06:650:115 mover_out 0**
**00:00:06:650:115 movel_out 0**
**11.** 00:00:07:000:029 LIGHT_IN DARK
**00:00:07:200:122 mover_out 1**
**00:00:07:200:122 movel_out 1**
**12.** 00:00:07:500:031 LIGHT_IN DARK
**13.** 00:00:08:000:020 LIGHT_IN ALL_DARK
**00:00:08:050:112 mover_out 0**
**00:00:08:050:112 movel_out 0**
**14.** 00:00:08:500:021 LIGHT_IN DARK
**15.** 00:00:09:000:028 LIGHT_IN DARK
**16.** 00:00:09:300:027 START_IN STOP
**00:00:09:300:126 mover_out 0**
**00:00:09:300:126 movel_out 0**

The results of this simulation were found identical within a reason for both the Linux and the bare-metal version.

After running various scenarios to verify the model behavior on the board, the driver interfaces were mapped with the robot sensors and actuators. The START_IN driver is attached to a button for starting/stopping the robot, and acts as an active device in this case. The LIGHT_IN driver is associated with a reflectance sensor for sensing the surface brightness and acts as a passive device since polling is needed to collect the sensor values. The output drivers MOVER_OUT and MOVEL_OUT are connected to two servomotors. The same behavior was observed, and the robot followed the line as expected. It is essential to emphasize here that the same models were used in both the Linux and the bare-metal versions. Only drivers had to be adapted. Videos for the Lego[1] and STM32-based[2] robot are available.

# 6. CONCLUSION

A new version of E-CD++ was presented. This version allows E-CD++ to run on bare-metal. It also provides a DEVSRT based execution engine that acts as a microkernel while models behave like processes. The main purpose of the new version was to have an OS independent platform that would be fully portable and loadable onto various development boards by removing its Linux dependency. The required system calls have all been replaced with implementation specific to the needs of E-CD++ and the system has proven to execute on target platforms.

A Line Tracking Robot example was developed and E-CD++ used throughout the entire development. The system was decomposed into several atomic and coupled models connected via a well-defined hierarchical scheme, where simply the Robot consisted of three main components: Sensor Unit, Control Unit and Movement Unit. The Sensor Unit receives light reflection readings then sends them to the Control Unit. The Control Unit analyses the data received and determines in whether the Robot is on a valid path or not and sends movement signals to the Movement Unit accordingly. The Control Unit commands the Movement Unit to

either move forward, turn or stop. To illustrate our approach, the Sensor Controller was modeled according to DEVS formal specification. Then, a corresponding implementation was presented. Finally, a simulation of the implemented model was run using ECD++. Tests were carried in both virtual and real environments. The results were satisfactory and followed the models' specification.

Based on the case study results, it can be seen that the implementation of E-CD++ as stand-alone software provides results that are identical to those of the simulated system. The case study has demonstrated the full range of abilities of E-CD++ through the use of internal and external transitions, as well as the execution of output functions and the interaction with hardware devices. Through the use of this tool, the simulation and implementation phases are linked as the initial models are deployed on the target hardware. We were also able to execute the engine on bare-metal on different ARM-based boards without the need of middleware RTOS.

The current version still needs to be ported onto a broader variety of platforms. Although the main modules are easy to port onto new platforms, the user will need to find appropriate drivers for the desired platform and be familiar with low-level programming. One of our future objectives is to provide a set of libraries and drivers for multiple microcontrollers to ease development task and only require the definition of DEVS models.

Another goal is to explore IoT applications. Indeed, provided hardware with connectivity capabilities, input/output ports can be associated with a network instead of traditional sensors therefore allowing I/O to be received and sent from any connected device. For instance, data could be sent by the DEVS-based kernel to a cloud based simulator or to any other connected hardware. Our execution engine could be used to connect small data and big data, and build diverse IoT applications.

## 7. REFERENCES


[1] Advanced Real-Time Simulation Laboratory. 2013. Line Tracking Robot on Lego Hardware. Video. Retrieved March 1,2015 from *https://www.youtube.com/watch?v=mTtlSV7WbuI*

[2] Advanced Real-Time Simulation Laboratory. 2015. Line Tracking Robot on STM32 (Early Debug Version). Video. Retrieved March 1,2015 from *https://www.youtube.com/watch?v=X2itlznkoVw*

[3] Bergero, F. and Kofman, E. 2010. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 87, 1-2 (2010), 113-132.

[4] Cho, S. M. and Kim, T. G. 1998. Real-Time DEVS Simulation: Concurrent, Time-Selective Execution of Combined RT-DEVS Model and Interactive Environment. In *Proceeding of 1998 Summer Simulation Conference* (Reno, NV, USA, July 19 - 22). SCSC '98. Society for Computer Simulation International, Vista, CA. 410-415.

[5] Cho, Y., Hu, X., and Zeigler, B. P. 2003. The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems. *Simulation* 79, 4 (2003), 197-210

[6] Edwards, S., Lavagno, L., Lee, E. A., and Sangiovanni-Vincentelli, A. 2001. Design of embedded systems: formal models, validation, and synthesis. In *Readings in hardware/software co-design*, De Micheli G., Ernst R. and Wolf W. (Eds.). Kluwer Academic Publishers, Norwell, MA, USA, 86-107.

[7] Furfaro, A. and Nigro, L. 2009. A development methodology for embedded systems based on RT-DEVS. *Innovations in Systems and Software Engineering* 5, 2 (2009), 117-127.

[8] Hu, X., Zeigler B.P. and Couretas J. 2001. DEVS-On-A-Chip: implementing DEVS in embedded java on a tiny internet interface for scalable factory automation. In *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference* (Tucson, AZ, USA, July 14 - 18, 2001). IEEE, New York, NY, 3051-3056.

[9] Hu, X., and Zeigler, B. 2004. Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example. *Journal of Intelligent and Robotic Systems* 39, 1 (2004), 71-87.

[10] Hu, X., and Zeigler, B. 2005. Model Continuity in the Design of Dynamic Distributed Real-Time Systems. *IEEE Transactions on Systems, Man, and Cybernetics Part A* 35, 6 (2005), 867-878.

[11] Li, Q., and Yao, C. 2003. *Real-Time Concepts for Embedded Systems*. CMP Books, San Francisco, CA.

[12] Moallemi, M., and Wainer, G. 2013. Modeling and simulation-driven development of embedded real-time systems. *Simulation Modelling Practice and Theory*. 38, 0 (2013), 115-131.

[13] Sloss, A., Symes, D., and Wright, C. 2004. *ARM system developer's guide*. Elsevier/ Morgan Kaufman, San Francisco, CA.

[14] Wainer, G. 2002. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience.* 32, 13 (November 2002), 1261-1306.

[15] Wainer, G. A. 2009. *Discrete-event modeling and simulation: a practitioner's approach*, CRC Press, Boca Raton, FL.

[16] Wainer, G. A., Glinsky E. and MacSween P. 2005. A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism. In *Model-Driven Software Development*, Beydeda S., Book M. and Gruhn, V. (Eds.). Springer, Berlin, Heidelberg, 363-383.

[17] Wainer, G. and Castro, R. 2011. DEMES: a Discrete-Event Methodology for Modeling and Simulation of Embedded Systems. *Modeling and Simulation Magazine*. 2 (April 2011), 65-73.

[18] Yu, H. Y., and Wainer, G. 2007. eCD++: an engine for executing DEVS models in embedded platforms. In *Proceedings of the 2007 Summer Computer Simulation Conference* (San Diego, CA, USA, July 15 - 18, 2007). SCSC '07. Society for Computer Simulation International, Vista, CA, 323-330.