Simulation

Methodology

Simulation: Transactions of the Society for Modeling and Simulation International
2021, Vol. 97(7) 485–506
© The Author(s) 2021
DOI: 10.1177/00375497211003130
journals.sagepub.com/home/sim
SAGE

A DEVS-based engine for building digital quadruplets

Daniella Niyonkuru and Gabriel Wainer

Abstract
Development of Embedded Real-Time Systems is prone to error, and developing bug-free applications is expensive and no guarantees can be provided. We introduce the concept of Digital Quadruplet which includes: a 3D virtual representation of the physical world (a Digital Twin), a Discrete-Event formal model of the system of interest (called the "Digital Triplet"), which can be used for formal analysis as well as simulation studies, and a physical model of the real system under study for experimentation (called the "Digital Quadruplet"). We focus on the definition of the idea of a Digital Quadruplet and how to make these four apparati consistent and reusable. To do so, we use the Discrete-Event formal model as a center for both simulation and execution of the real-time embedded components with timing constraints, as well as a common mechanism for interfacing with the digital counterparts, providing model continuity throughout the process. Here we focus on a principal part of the Digital Quadruplet idea: the provision of an environment to allow models to be used for simulation (in virtual time), visualization, or execution in real-time. A Discrete-EVent Systems specifications (DEVS) kernel runs on bare-metal hardware platforms, avoiding the use of an Operating RTOS in the platform, and the combination with discrete-event modeling engineering.

Keywords
Digital Twin, Digital Quadruplet, DEMES, Discrete-Event Modeling of Embedded Systems, cyber-physical systems, DEVS, real-time DEVS

1. Introduction

Embedded Systems are computing systems with tightly coupled hardware and software, which are designed to perform specific functions. Embedded systems have paved the way for a world of connected devices in areas such as our homes, workplaces, automobiles, medical care, and unmanned aerial vehicles. They are ubiquitous, diverse, and can be found in numerous industries (Aerospace, Consumer Electronics, Defense, Medical Equipment, and Transportation). A special category of Embedded Systems, referred to as Real-Time Embedded Systems (RTES),[1] needs to respond to external events with strict timing constraints. RTES are not only subject to functional and logical correctness, like other software systems, but they also must deliver results within strict timing constraints. Missing these deadlines may lead to significant loss and, in some cases, catastrophic consequences. For instance, in an airbag deployment system, a delay could lead to serious injuries or even death. Besides, RTES usually operate in limited-resource environments, are required to have a small memory footprint, limited processing capabilities, and low power consumption. RTES software should be designed to meet these requirements, as well as varied hardware/software interfaces as well as scalability and complexity.

Despite these stringent requirements, there is still no well-established and widely accepted robust framework for designing RTES. Software is the most expensive and least reliable part of RTES. The deficiencies mainly come from the need of complex development cycles and the system verification facilities.[2] On the one hand, disruptions exist through the development lifecycle because different artifacts and tools are used for analysis, design, test, and implementation.[3] Indeed, some tools/methods are used in each stage of development, while others are better for other development stages. For instance, MATLAB may be used to build mathematical models to analyze the system

Systems and Computer Engineering, Carleton University, Canada

Corresponding author:
Gabriel Wainer, Systems and Computer Engineering, Carleton University, 1125 Colonel By Dr., Ottawa, ON K1S5B6, Canada.
Email: gwainer@sce.carleton.ca

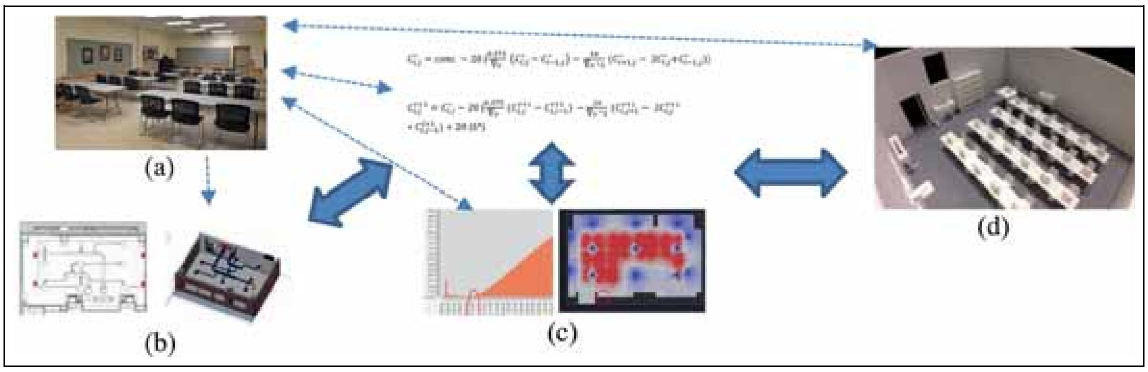

Figure 1. (a) Real System (University Classroom); (b) Digital Twin (Building Information Model – BIM – of the University Classroom under study); (c) Digital Triplet (Equations of the spread of CO_2 in the classroom); (d) Digital Quadruplet (replica at scale of the classroom, which can be wired with sensors and actuators for experimentation).

behavior in the analysis stage. However, these mathematical models are scarcely used during the software design stage, where software models (for instance, models designed using Unified Modeling Language (UML)) are more popular. Similarly, software models are not adequate for system behavior analysis; as well, none of these models can be directly executed on the target hardware platforms, and consequently these models are abandoned during the implementation phase in favor of traditional techniques (for instance, methods based on programming languages such as SystemC). These traditional techniques are nevertheless unable to handle the uprising heterogeneity. They rather result in increased design rigidity and complex software that can no longer guarantee sufficiently reliable systems. On the other hand, RTES system verification is particularly complex. Since these systems are a tight mix of embedded software and hardware components that are required to respond to external events in real time, correctness is hard to achieve.

In recent years, formal methods for RTES design[4] have shown promise in dealing with these issues, but they do not scale up well. Current trends investigate model-based design techniques to cope with heterogeneity, but the lack of formal modeling methods and the existence of some inconsistencies make RTES development become an ad hoc process that is expensive, time-consuming, and error prone. Instead, a practical solution to this problem is to use Modeling and Simulation (M&S) techniques for testing conditions regardless of the application's size. In particular, the use of Formal M&S provides a means for solving the problems stated above by combining the advantages of a M&S-based approach with the rigor of a formal methodology.[5]

Generally, M&S-based development lifecycles involve Requirements and Specifications (Requirements are outlined, and specifications determined), Modeling and Simulation (models of sensors, actuators, controllers, and other components are designed, and the verification is done through simulation). Then, a phase of Model Mapping is executed (models are mapped to software and hardware), and finally, a Prototyping and Implementation phase is carried out (models are executed in a real-time environment and incrementally replaced with their hardware surrogates).[6] Our long-term research goes beyond these concepts and focuses on what we coined as ''digital quadruplets.'' A Digital Twin[7,8] normally includes a visual representation of the physical system under study. We propose to expand such definition to include a formal model of the real-world system that we call the ''Digital Triplet.'' This is a formal model that can be used for formal analysis and can be used for simulation studies. In parallel, we propose to build a Physical Model, called the ''Digital Quadruplet,'' which is a replica of the real system under study at scale. The Quadruplet can be used for experimentation on a physical model resembling the actual real-world system under study.

Figure 1 shows the Digital Quadruplet concept. Starting with a real-world System of Interest (in this case, a classroom), we build a Digital Twin with all the objects in the building: walls, vents, windows, power, and lighting fixtures (including, for instance, CO_2 sensors to detect occupancy). The Digital Twin information is used to build a formal model: the Digital Triplet (which represents occupancy of the room, movement of students, activation of the CO_2 sensors, which, depending on the number of students, can be used to control the lights and HVAC levels in the classroom). We can use the Digital Triplet to analyze the behavior of students and generation of CO_2 in the classroom, as well as the Real-Time embedded controller using a formal model, as well as automating simulation studies of the overall system. The Digital Quadruplet is completed with a physical model of the classroom at scale, in which we install different kinds of hardware devices to conduct experimentation on a replica of the system of interest (for instance, we can install CO_2 sensors and generate CO_2 to study the behavior of the control system for the actual building, as well as experiments on fires, evacuation of the classroom, or reduction of carbon emissions with no risk and at reduced cost).

Our research focuses on how to make these apparati consistent and reusable by using formal models for modeling, simulation, execution of the real-time embedded components with timing constraints, and enabling visualization while providing model continuity throughout the process. The rest of the article describes a major part and contribution in this concept: the definition of a method and a prototype environment to build formal models that can be used for simulation (in virtual time) and visualization, or execution in real-time (i.e., steps (b) and (c) in Figure 1).

This kind of M&S-driven development approach must also ensure the progression from M&S to Prototyping and Implementation. The models created for simulation are often not reused, and the implementation is normally done from scratch or using a different methodology.[9] Instead, a formal methodology providing model continuity can help to solve this problem. Model continuity is defined as the ability to retain the same models that were initially developed, throughout the various stages of the software development lifecycle. Several 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 it more difficult to prove valuable properties about the models under development), and they do not provide model continuity in the RTES development lifecycle.[5] DEMES (Discrete-Event Modeling of Embedded Systems), an M&S-based development methodology based on discrete-event systems specification, offers a practical approach with a formal rigorous method in which models are consistently used throughout the development cycle.[10]

DEMES is an M&S-driven methodology that uses the DEVS formalism[11] to provide a formal foundation to tackle the issues of RTES development discussed above. The DEMES concept has been applied to the development of different tools that include the E-CD++ environment, offering a unified and consistent development environment. E-CD++ provides seamless integration from modeling to development and it reduces the design efforts by allowing the same models defined for simulation analysis to be reused in real-time in the hardware target platform. The methods and tools are based on the DEVS formalism, and it provides a mechanism to run the models on a target hardware without modifications, allowing modeling reuse and evolvability. DEVS models can be tested through simulation and then deployed in the target platform, through incremental deployment in the target hardware platform.

In Moallemi and Wainer,[5] we introduced E-CD++, an environment using a DEVS real-time executive running on top of the Xenomai real-time kernel. Xenomai enabled the manipulation of the underlying hardware platform. Nevertheless, this approach has problems: to run E-CD++, Linux and Xenomai, high-performance microcontrollers are needed (including powerful processors, memory, and in many cases, secondary memory to allow the software stack to be executed without any problems). Furthermore, the main constraint is that the OS introduces unacceptable latencies and unpredictable execution times due to the introduction of an abstract layer implementing the OS services that one cannot control. To prevent these issues, we have introduced a new method for bare-metal execution, with a real-time kernel that does not rely on Xenomai services or the Linux OS. The objective is to be able to execute models directly on the target system hardware without the need of an OS. The new E-CD++ presented here provides functionalities like those of a real-time kernel, while DEVS models operate as system processes. One of the advantages of this approach is the reduction in the memory footprint, as the models run directly on bare metal only including the necessary drivers and the DEVS kernel. We have moved from general purpose processors with at least 1Mb of RAM and 64Mb of Flash (needed to store the kernel of the OS and the DEVS environment, plus the models running) to microcontrollers with 256Kb of RAM and 16Mb of Flash memory, being able to run more complex models thanks to the space freed by the OS, which is now not needed. Simultaneously, running on the bare hardware provides complete control on schedulability of the tasks being executed and better analysis and control of the latency during the input/output activities with the target hardware. The use of an OS provides a layer of abstraction that makes development easier, but also introduces expensive memory requirements as well as adding overhead in using the central processing unit (CPU) and the scheduling algorithms can introduce unexpected effects in the input/output processes.

The research presented here extends the applicability of M&S-driven development by providing a DEVS execution engine independent from the OS. As a result, target devices such as low power microcontrollers are now covered. Using the methodology, we can build digital quadruplets, including advanced DEVS models for analysis and simulation which can be integrated with visualization environments and executed in a hardware surrogate without modifications. Likewise, we are not constrained by limited resources and performance barriers introduced by the OS. To show the feasibility of the approach, we present a case study using the bare-metal environment.

The rest of the paper is organized as follows. Section 2 presents a brief overview of the DEVS formalism and its application to real-time embedded systems design. Section 3 discusses E-CD++ architecture and its implementation as an embedded software. Section 4 describes a case study and the definition of a Digital Quadruplet, focusing on the software development of an embedded component using E-CD++ and the model definition and execution.

2. Background

With the rising complexity, scalability, and heterogeneity of RTES, alternative approaches are needed to cope with these considerations. Model-based design methods offer solutions but are still limited to the early stages of development or only cover a limited range of devices.

2.1. Designing RTES

Since RTES are partly made of hardware and software, three important design aspects have to be considered: the hardware design, the co-design of hardware and software, and the design of embedded software. The implementation directly on hardware provides the advantage of better energy efficiency; however, these kinds of implementations are very expensive and generally require long design times.[12] Although these methods are appropriate for small systems, they are fast becoming infeasible due to the ever-increasing heterogeneity and complexity of embedded applications. For this reason, we need to raise the design to a higher level of abstraction and provide an integrated approach to software and hardware design to support the new system.[13]

Traditional design divides hardware and software design to conquer them separately. Indeed, most embedded systems are designed from a register level description for the hardware part on one hand, and the embedded software code on the other hand.[14] The implementation is obtained using a top-down approach.[15] Hence, traditional design methodologies trace their origins from either software or hardware traditions[2,16] but do not cover hardware–software co-design. Co-design[17] is essential in order to design complex applications since hardware components are diverse in heterogeneous systems and software–hardware interface should be handled earlier in the development cycle.

Two types of methodologies have emerged from traditional design: language-based (software-centric) and synthetic-based (hardware-centric) methods. Language-based methods are centered on a specific programming language with a particular target runtime system. C and RT-Java are such examples. Synthesis-based originates from hardware design techniques. The development starts with a system description, usually structural, in a tractable fragment of a Hardware Description Language (HDL) like VHDL and Verilog.[2] Furthermore, with this type of design, system designers can write systems at the system level using C and C ++.[18] The C/C++ description is refined; and then manually translated into synthesizable HDL by designers: an error-prone step.[19] Consequently, the usage of different languages and environments leads to increased inconsistency, hardly verifiable, and implementation platform dependent system specification. The rise in heterogeneity and complexity of actual systems requires a generic tactic: a modular, component-based approach that addresses heterogeneity and complexity at higher levels, integrates both hardware and software design, enables verification, and offers a unified and consistent development environment.

Methods centered on the semantics of abstract system description were later introduced in an attempt to gain independence from specific implementation platforms. They combine both language and synthesis-based techniques in order to enable hardware/software co-design. Examples include SystemC,[20] which combines synchronous hardware semantics with asynchronous execution mechanisms from C++. Recent methodologies are execution semantics independent, offer higher levels of abstraction, and go beyond platform independence. They are built on modeling languages such as the UML[21] and Architecture Analysis and Design Language (AADL).[22] These methods attempt to provide genericity both in the choices of implementation platform and semantics for abstract system descriptions and preserve hardware–software co-design.

Model-based design and development seek to address heterogeneity, and targets the system's increasing scalability and complexity early in the development cycle by using models to describe the system. Therefore, RTES design consists of transforming a more abstract model into a less abstract one, until a final model is obtained and ready to be deployed. Different modeling methodologies have been introduced in literature and include UML-RT, ECSL, Ptolemy II, and Behavior, Interaction and Priority (BIP). While most model-based methodologies solve existing challenges in the design phase, they do not cover the full spectrum of embedded system development, lack the needed formalism, and often remain limited to the early stages of development.

For example, while UML-RT (the Unified Modeling Language for Real-Time) uses UML extended to real-time to represent special characteristics of RTES and can be used to construct software design models, it is not adequate to implement formal verification and hardware/software co-design. As a consequence, UML models must be translated into or used along with some formalism. For instance, in Murillo et al.,[23] UML/MARTE is mapped to a mathematical formalism; in Nascimento et al.,[24] the functional behavior of a UML model is translated into a network Labeled Timed Automata (LTA) for formal verification. In addition, UML lacks the methodology to complement the design implementation after requirement specification; it also has consistency problems, which is a major set-back for this design methodology.[21]

Another example is Matlab/Simulink, a popular commercial tool used for M&S of embedded systems. The tool is based on data flow languages' semantics that are defined through a simulation engine using block diagram models to describe an analytical model. A powerful graphical interface is available for visual construction and integration of hardware blocks. Simulink can be integrated with several tools, such as Stateflow, Simulink Coder, and Embedded Coder for event-based modeling, physical modeling, and code generation. Simulink is mainly used for simulating real-time systems and does not cover the entire RTES development spectrum. Although the code generation tool produces C/C++ code for embedded processors, the generated code has limited usage and does not support all the functionalities of the Simulink blocks.[25] MathWorks brought in Simulink Code Inspector for code verification. However, it is limited to a subset of Simulink and Stateflow capability and can fail to detect model incompatibilities. On the other hand, ECSL (Embedded Control Systems Language), a graphical modeling language built using the Generic Modeling Environment (GME), was designed in the context of embedded automotive systems to extend Matlab with capabilities such as requirements specification, verification, mapping onto a distributed platform, scheduling and performance analysis. However, formal proof of correctness of the resulting application behavior remains an open problem and testing techniques are still under investigation.[26] Ptolemy II, a structured and hierarchical approach, attempts to attack heterogeneity by using a specific model of computation (MoC) that defines how computation takes place among a structure of computational components and covers the flow of data and control.[9,27] BIP (Behavior, Interaction, and Priority) is methodology for modeling heterogeneous real-time components and defines them as the superposition of three layers: Behavior (a set of transitions), Interactions (between transitions), and Priorities (to choose among interactions). This approach preserves properties during model composition and supports analysis and transformation across different boundaries.[28]

Model transformation is central to all model-based approaches and must preserve essential properties. Some transformations are automated, and others are guided by the designer.[16] Automation uses code generators (such as done in Matlab/Simulink,[25] UML-VHDL[29,30]) which often produce inefficient code from equation-based methods and complicate verification. The other alternative is to develop high-level programming languages able to express reaction constraints together with the satisfaction of reaction constraints guarantee.

Although model-based approaches handle modern systems' complexity and heterogeneity by raising the level of abstraction and allowing a hardware–software co-design, research remains to be done in the areas of the development cycle and system verification. Indeed, direct model continuity should be supported and efficient model transformation provided to ensure that initial models are reused through the development cycle, maintain consistency and offer a unified development framework. In addition to being effective at the high level (system description), model-based methods should also be effective at lower levels (implementation) and be applicable to devices with limited resources (e.g., memory) as well as large systems in order to compete with traditional methods. Formal methods can help with system verification, allowing demonstrating correctness for certain system properties.

To meet the previous considerations, a formal methodology that provides model continuity can be applied. In the following sections, we will introduce DEMES,[10] a formal M&S-based development methodology based on Discrete-EVent Systems specifications (DEVS).[11,31] DEVS is a well-defined formalism that is expressive, operates at a high level of specification, and can be used to represent both computing systems and the physical systems they control. DEMES provides a formal approach where models are used throughout the development cycle.

2.2. Using DEVS as a formal M&S-based approach for RTES design

DEVS is a Discrete-Event M&S formalism for dynamic systems. The DEVS formalism decomposes complex system designs into basic models called *atomic* and composite models called *coupled models.*[11] It follows an exact set of rules for defining modeled system state changes for input events or time delay triggers. DEVS is particularly suitable for RTES because it provides a rich structural representation of the components and a formal means to explicitly specify their timing which is the center of the real-time system. It has been proven to be successful in different complex systems.[32,33]

Besides, DEVS is the most general Discrete-Event Formalism, and most existing Real-Time techniques (e.g., State Charts, VHDL, Verilog, Timed Automata) can be expressed as DEVS. Plus, DEVS theory does not only provide a rigorous methodology for model construction but also proposes an abstract algorithm independent of the underlying hardware and middleware.

a) The DEMES development cycle. DEMES focuses on bridging formal methods, modeling, simulation, and real-time execution to develop real-time applications as well as for studying their interaction with the physical environment. The objective is to enable the original formal models to become a part of the final application deployed in the target hardware platform. This is accomplished by using M&S at an early stage during the design process and gradually replacing the M&S model with hardware surrogates and new software components that implement the original model without changes, in incremental fashion and providing model continuity. After thorough testing on the simulated platform, the model can be incorporated into the

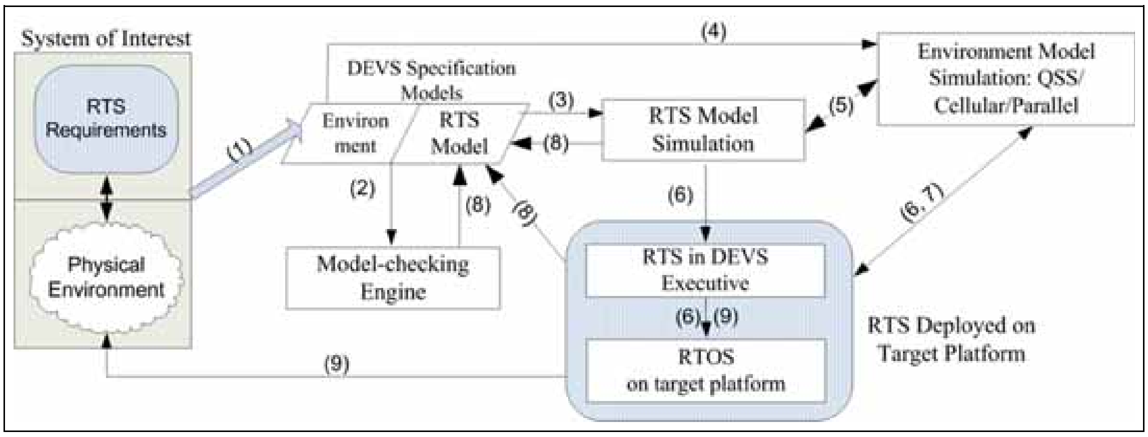

Figure 2. Discrete-Event Modeling of Embedded Systems (using DEVS).

target environment and reused throughout the design process.

Figure 2 shows the architecture of the DEMES methodology. A designer starts (1) by modeling the System of Interest (a real-time system and its environment) using **formal specifications** (DEVS or alternative techniques such as Statecharts, Hybrid Automata, or Modelica). These models are converted into a DEVS representation, then transformed into alternative representations (for instance, using Timed Automata[7]), and then verified using **model-checking** tools (2). The formal verification phase can take a long time: the rules applied to check the models can be complex to define, and in some cases, the model-checking engines produce explosion of states, making the formal verification process complex. In those cases, we can take advantage of the formalization in DEVS, and the same models used for model-checking can be used to test the components in a simulated DEVS environment (3). Another advantage is that in this process we can also simulate physical environment in which the application is deployed (4) and combine it with simulations of the RTS model under particular loads of interest to the application (5). In these simulation studies, instead of obtaining general answers for all the possible cases of execution (like those provided by model-checking), we can simulate individual behaviors of the different submodels under specific conditions. Simply put, system properties can be studied formally using model-checking, and the proofs complemented using M&S. It can also be used for collaborative hardware/software design. This process includes also modeling the physical system in which the software is embedded. We model the physical system characteristics that the RTS is interacting with and controlling, and they can be simulated to study how the changes in the physical system affect the software. Subsequently, the tested submodels can then be deployed in phases to the target platform. (6). A **real-time Executive** executes the models on the particular hardware. If hardware is not readily available, software components can be developed in stages, tested against hardware models to verify feasibility, and make design decisions early. As the design process progresses, both the software model and the hardware model can be refined (in a spiral or tornado life development cycle), gradually setting up checkpoints with actual prototypes. The executive allows to execute dynamic models and to schedule static and dynamic tasks. At this point, parts that are not verified in formal and simulated environments are tested. Most of the testing phase (7) can be done using **simulation** (with faster than real-time performance), even if the hardware is unavailable. Simulation provides a risk-free testing environment, and will be applicable in cases where real-life testing is impossible due to risks, ethical, or practical issues. With DEMES, design changes are done incrementally (8), providing a consistent set of apparati throughout the development cycle. The cycle ends with the RTS fully tested and every model deployed on the target platform.

This approach has various advantages when compared to other methodologies. For instance, the methods discussed in section 2.1. were semi-formal and do not provide direct model continuity. In Huang and Sarjoughian,[33] the authors presented a comparison between DEVS and UML-RT showed that features such as time, scheduling, and performance coded using UML constructions are not formally defined. Instead, formal modeling techniques such as DEVS provide sound syntax/semantics for structure, behavior, time representation, and composition, and are useful for explicit computation.[3,5] In addition, the DEMES approach provides the following advantages:[6,12]

- Reliability: logical and timing correctness is based on DEVS, which has a strong system theoretical root and sound mathematical theory.

- Model reuse: DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition, supported by the formal concept of closure under coupling.
- Hybrid modeling: 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, which allows knowledge reuse.
- Process flexibility: hybrid modeling capabilities are transparent for the executive, which is defined by an abstract mechanism that is independent from the model itself.
- Verification and validation: the definition of experimental frames can be partially automated and formal verification is possible. Formal verification will be discussed in the next section.

b) DEVS model verification for real-time embedded applications. One important aspect of DEMES is the use of model-checking, which allows the designer to verify the model's correctness and eventually produce formally correct software. Therefore, deployed systems will have very high reliability as the formal verification permits error detection at the early stages of the design. Informal techniques usually rely on extensive testing; they can reveal errors but are unable to prove the nonexistence of errors. Instead, formal techniques can prove the correctness of a design. The main drawbacks, however, are that formal techniques are constrained in their applications (not easy to scale up), and are usually applied to abstract models and not the final executable system.

When models used for M&S are formal, their correctness is verifiable. Another advantage of executable models (such as the ones featured in DEMES) is that they can be deployed to the target platform, thus providing the opportunity to use the controller model not only for simulations but also as the actual code executing on the target hardware. Thus, the verified model is itself the final implementation executing in real time. This prevents any new errors that might appear during the transformation of the verified models into an implementation, hence guaranteeing a high degree of correctness and reliability.

Hwang demonstrated that the verification of general DEVS is undecidable through reachability analysis.[34] Therefore, no model-checking tools have been built to verify models built-in DEVS. As a consequence, various DEVS extensions were proposed, in which the authors added constraints to the DEVS formalism to allow formal verification, in particular, by restricting the time-advance function to nonnegative rational numbers and the elapsed time used in the external transition function. RTA-DEVS[35] is one of those subclasses of DEVS that provides the modeler with a method that is expressive to model complex systems and can be verified through formal model-checking techniques. RTA-DEVS models are a class of rational time-advance DEVS that can be transformed into equivalent Timed Automata that can be used to formally verify the desired properties using model-checking tools like UPPAAL.

c) Existing DEVS-based environments. Recent research has focused on DEVS application to low-level applications commonly found in embedded systems consisting of computer hardware and real-time software. Current DEVS-based development environments for RTES include PowerDEVS[36], a method to model hybrid systems and carry out real-time M&S, and Action-Level Real-Time DEVS (ALRT-DEVS)[37] in which Network-on-Chip systems are modeled and simulated in real time. In Hu and Zeigler,[3] the authors show how model continuity can be used in the design of dynamic distributed Real-Time systems. In Song and Kim,[38] we can see an application of the DEVS framework to the design and safety analysis of a RTES (a railroad crossing control system). In Moallemi et al.,[39] model reuse and interoperability were shown by interfacing E-CD++ and PowerDEVS. A System-On-Chip FPGA implementation of E-CD++ was presented in Moallemi and Wainer,[40] and a M&S-based design of embedded controllers on network processors in Castro et al.[41]

Compared to traditional methods, there are significant platform limitations: in Hu and Zeigler[3] or Furfaro and Nigro,[32] where the platform uses JAVA, the target platform should support the Java-DEVS real-time execution environment. PowerDEVS requires a Linux RTAI kernel.[42] In Moallemi and Wainer,[5] the DEVS-based tool depended on Xenomai/Linux kernel services. From the foregoing, although the DEMES approach offers multiple benefits, tools have to be improved to overcome limitations and support different hardware.

Earlier versions of E-CD++ relied on a variant of the Linux kernel. In Castro et al.,[41] E-CD++ was embedded on the core processor of an Intel IXP2400 Network Processor that runs RT Linux. It ran in the Linux User Space and used Linux Kernel services. In Moallemi and Wainer,[40] a configurable Linux kernel was downloaded to the SDRAM memory blocks on the AP1000 FPGA board. This dependency also included the use of the Xenomai real-time framework for Linux. Xenomai, installed as an abstract RTOS core, provided hard real-time functionality to the Linux kernel.

By eliminating OS and virtual machine restrictions, reducing the footprint of embedded applications, and bridging the gap between M&S and implementation increasing efficiency can be improved. Eliminating the virtual machine provided by the OS or the Java engines allow better predictability and improved control of the software

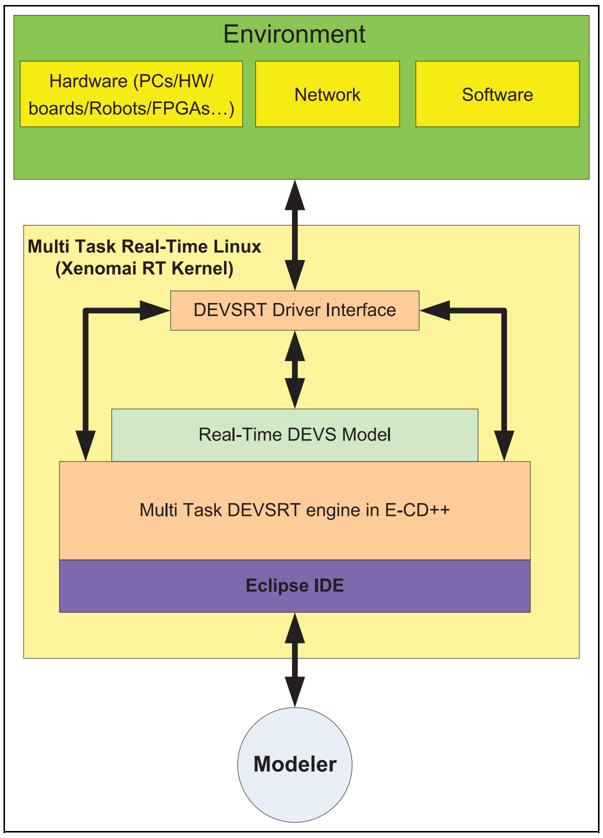
Environment
Hardware (PCs/HW/ boards/Robots/FPGAs…)
Network
Software
Multi Task Real-Time Linux (Xenomai RT Kernel)
DEVSRT Driver Interface
Real-Time DEVS Model
Multi Task DEVSRT engine in E-CD++
Eclipse IDE
Modeler

Figure 3. E-CD + + Layers.

components, allowing the reduction of latencies and permitting direct control on the hardware (which is not possible when an abstraction of the hardware is provided by an OS or similar software). In this case, all the developers use is DEVS and its well-defined functions (which includes a simple API with four functions). In the following sections we will discuss how this has been done, and will present E-CD ++ software components and its implementation.

2.3. *E-CD++*

Our RT kernel, E-CD++, allows the developers to follow the DEMES methodology explained earlier to design complete applications using a discrete-event modeling approach. E-CD++ was developed using our previous experience with the CD++ simulator[31] (a DEVS-based framework for M&S) and the RT-CD++ prototypes[43,44] (an extension of CD++ for real-time model execution). E-CD++ uses the original ideas for RT execution of DEVS models that uses the CD++ virtual time-advance function within a real-time context and provides an RT M&S platform for simulation-based verification of such models. Figure 3 shows the architecture of the E-CD++ framework, which allows the different stages in the DEMES process to be integrated in a seamless fashion.

For the models to be replaced directly with external entities, the I/O ports used for the DEVS models in E-CD++ implement the interface mechanism originally proposed for DEVSRT in the Driver Interface layer. The intermediate software used for this is a real-time kernel, and objects are imported into this platform as RT tasks at runtime. The user models, the driver objects, and the E-CD++ core objects are merged and compiled to produce an executable that runs on Xenomai. An Eclipse IDE allows for the graphical development of models (a graph-based representation can be used to specify the model's hierarchy, interconnections, and behaviors to automate model generation). The DEVS M&S algorithms allow for parallel execution of concurrent events through the implementation of a messaging behavior for model interaction. The Flattened Coordinator technique[45] improves the efficiency of the DEVSRT messaging behavior 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. The tool supports two execution modes: simulation and real-time. In simulation mode, virtual time advances using discrete-event algorithms. In real-time mode, the execution is driven by the real-time clock.

For E-CD++ to execute the tasks as outlined in the various layers described in Figure 3, four software components are needed (shown in Figure 4). These include: the Main Runtime Subsystem, the Modeling Subsystem, the RunTime Subsystem, and the Messaging Subsystem. The Main Runtime Subsystem manages the overall aspects of the real-time execution and provides timing functions with microsecond precision. This is the first object (created in a non-real-time context), which launches the Runtime Subsystem. It first registers atomic components, the Top coupled component ports that are connected to the external environment, reads in external events (if any, for testing purposes, from an existing event-file), and builds an external event table. After, it reads in the model file and builds the model hierarchy. Finally, it spawns the main real-time task in which the Root Coordinator is created to start the DEVSRT execution cycle.

The Runtime Subsystem consists of Simulators, Coordinators, and the Processor Admin, which drive the execution of the models. In E-CD++, the Simulators work on runtime engines that correspond to atomic components, and they perform the main job of executing the internal transition and output functions for each of the submodels, following DEVS algorithms. The Root Coordinator manages the real-time event scheduling. It initializes a Driver object that launches real-time input driver tasks (associated

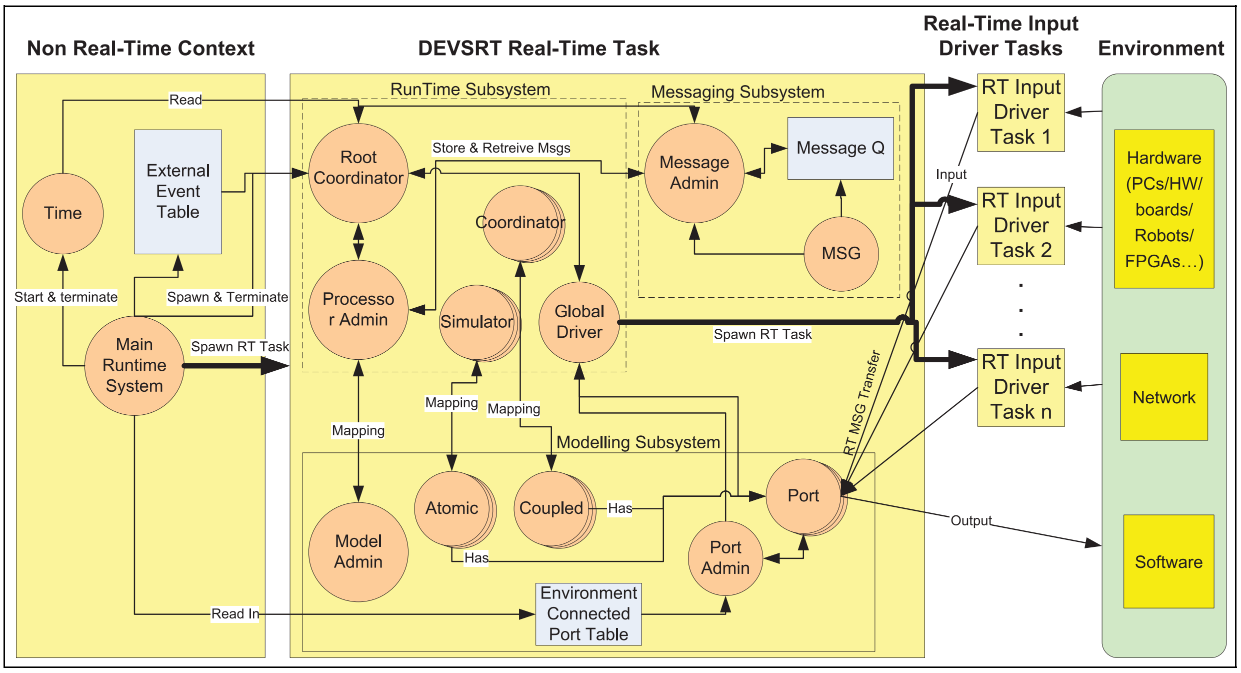

Figure 4. E-CD++ software components.

with input ports of the Top coupled component in the DEVS model hierarchy).

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, the Processor Admin within the Runtime Subsystem defines Coordinators and Simulators in order to manage the behavior of the model and drive its execution.

The Messaging the subsystem provides is in charge of transmitting messages between the different components in the Runtime Subsystem, which makes the model execution advance (in virtual or real time).

This implementation is closely dependent on the Linux OS and restricts supported devices. E-CD++ on Bare Hardware removed this limitation, provides a DEVS execution engine that resides in a different microcontroller, and it is OS independent, making it applicable to a broad variety of hardware. To achieve this objective, several changes were required and will be presented in the next section.

3. E-CD++ on bare hardware

As discussed in section 2, the development of embedded applications using E-CD++ requires several changes to the current iteration of the software. 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 to operate without OS support and directly interface with hardware devices.

3.1. *High-level design*

E-CD++ was built with the assumption that it would run on a variant of the Linux kernel. This imposes memory capacity, processing, and portability limitations. 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. Both implementations on the network processor and the FPGA board required a Linux kernel to be downloaded on the target. In Castro et al.,[41] E-CD++ was embedded on the core processor of an Intel IXP2400 Network Processor that runs RT Linux, while in Moallemi et al.,[39] a configurable Linux kernel was downloaded to the SDRAM memory blocks on the AP1000 FPGA board. This dependency also includes the use of the Xenomai real-time framework for Linux.

Porting E-CD++ to embedded platforms with small memory capacities is affected by the size of the application as well as the Linux kernel. To tackle this challenge, we could add external memory or use a network connection to interface with the target platform from a M&S platform. The inclusion of the Linux kernel and use of network drivers reduce the portability of E-CD++ between different platforms. In the design of the system, a target platform

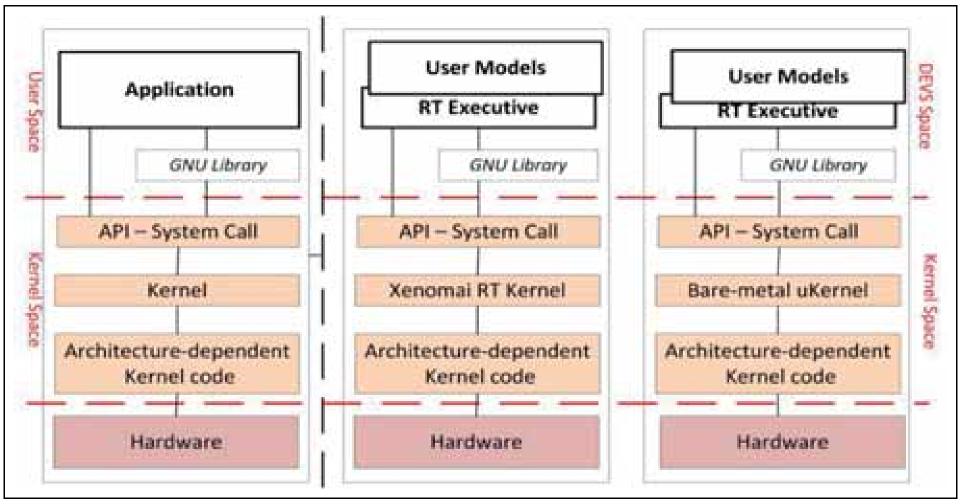

Figure 5. RTOS vs bare metal – an overview.

must be chosen based on the availability of a Linux kernel for that architecture as well as the availability of network drivers and a suitable API that allows for control of hardware devices over a network interface.

As a solution to the above challenges, we propose a new architecture to remove these limitations through the removal of Linux dependencies as well as the implementation of the Driver model to eliminate the need for API availability, as seen in Figure 5.

The middle part of Figure 5 shows the corresponding components for the Xenomai version of E-CD++. In this case, the application layer is made of user models and real-time executives in charge of executing them. These layers are in what we named the ''DEVS space'' as they pertain to the DEVS framework. The real-time executive relies on the Xenomai kernel. To switch to bare metal, several changes are needed, namely the real-time executive, the introduction of a microkernel – that provides functions to handle system calls, to manage memory and hardware resources – and the use of a small and optimized GNU library for embedded systems (in our case Nanolib) for code size reduction.

To successfully run on bare metal, all the Operating System dependencies needed to be removed and a microkernel developed to provide the essential services offered by the real-time kernel, including file, memory, and hardware resource management. Applications request OS services via system calls; for the bare-metal real-time executive we identified the system calls needed and provided functions with the same signature but with a re-designed implementation that takes into account the limitations and environment of the target platform. These include:

- file and input/output management to read the user-model files;
- memory management (e.g., *sbrk*) for allocation/deallocation of memory, for heap memory allocation;
- real-time calls to the Linux kernel, by interfacing with the hardware clock of the target platform;
- other services, providing similar functions to replicate them using hardware components (i.e., onboard memory and low power modes);
- startup code, low-level initialization, linker script, and interrupt handling mechanisms.

Many of the OS services are unnecessary (for instance, inter-process communications, as there is only a single process running in the DEVS kernel). Multi-processing can be implemented directly at the model level and natively handled by the execution module based on DEVS algorithms. In this context, models act as processes and the DEVS coordinators as the Process Scheduler. Periodic and aperiodic tasks can be managed with timers and interrupts. The new functions were developed to provide essential functionalities requested through system calls without the overhead of a full OS kernel. This provides the capability of scheduling function calls at timing intervals as dictated by the onboard timer without the need for a real-time framework to enforce these constraints. The DEVS kernel needed other updates, including a new implementation of a Driver model and the use of a Flattened Coordinator technique.[45]

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 M&S platform and the target platform as illustrated in Figure 5. The new functions were implemented to provide the same functionalities as the original system calls without the overhead of a full OS kernel.

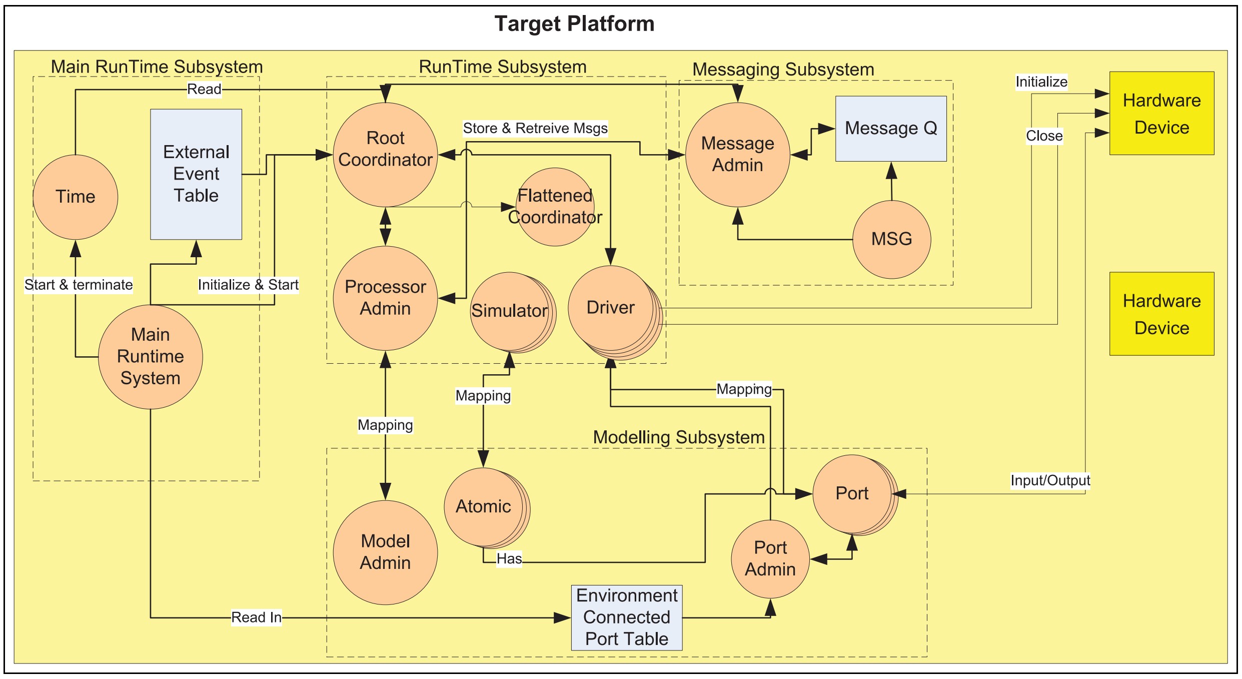

Figure 6. E-CD + + Software architecture.

In section 3.2, the main changes mentioned above will be detailed and their implementation further explained.

3.2. Implementation details

In this section we discuss a few details related to the implementation to show how the general ideas have been carried out. A complete implementation is available in open source (`https://github.com/SimulationEverywhere/RT-Cadmium-Models`). Initially, the Driver concept and the Flattened Coordinator were adapted to achieve our goals. These components are described in Figure 6. The Runtime and the Modeling subsystems now include a Flattened Coordinator as well as the integration with the Driver model. The Flattened Coordinator has been added to the Runtime Subsystem where it replaced the many Coordinators that are needed in a hierarchical Coordinator algorithm as described by Zeigler et al.,[11] which were previously used in E-CD++. The Driver has also been added to the Runtime Subsystem providing an interface to initialize hardware devices as well as to interact with the Ports that are associated with the model.

With the aid of the Driver object, external I/O devices can be controlled through the encapsulation of hardware specific functionality. Using this model, we can interface a wide range of devices and improve portability by implementing basic drivers that are accessed by E-CD++. From an implementation point of view, the Driver model manages hardware devices using two classes: Port and Driver. The Port class resides in the Modeling subsystem while the Driver class is in the Runtime 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. In the previous implementation, the Port class established API commands over a network interface but with the bare hardware implementation, the Port class includes Input and Output low-level functions that provide direct interface with the hardware device. In the case of Inputs, the receipt of a signal on a Port will cause the generation of a DEVS message, which is then added to a message queue processed by the Root Coordinator. When configured as an output, the Port will receive the data from a DEVS message from the Driver class, which will then be translated into a signal that can be interpreted by a hardware device. The bare-metal implementation allows using hardware and software interrupt service routines of the target platform to notify E-CD++ of I/O events. The specific hardware interrupts associated with the device can then be used to generate DEVS messages based on the values received. Similarly, software interrupts can be programmed to provide periodic polling.

We also needed to define two types of devices that a Port/Driver may be associated with: *passive* and *active* devices. Passive devices (for example, passive sensors) must be polled at specific intervals to determine their current state; hence, interfacing them requires the implementation of a periodic timer interrupt that requests the state of those devices. This can be accomplished through the creation of a software interrupt that is tied to a division of the base clock, allowing for the 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. 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. 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.

As embedded platforms are generally limited in terms of memory and a CPU power, the OS dependency of the C++ library was removed to streamline the performance of the software as well as increase the portability of the overall system. E-CD++ is the only application running on the target platform, reducing the memory footprint (as there is no OS running). The replication of some key OS functionalities is achieved using hardware devices available on the target platform (real-time clock, onboard memory, low-power modes, etc.).

The Main Runtime subsystem is the first object to be created. It oversees initializing the system timing. This is achieved with a 32-bit timer set to trigger at 1MHz therefore providing microsecond precision (as Xenomai does). A constraint is that the microcontroller's clock frequency should be at least 1MHz (which is common nowadays). Then, it registers the model file, the only file referenced by E-CD++ during execution on the target platform. Since we do not have a complete OS file system, we developed a pseudo file system to maintain continuity between desktop simulation and target real-time execution. To mimic this behavior, 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 that 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.

One of the other major tasks of the Main Runtime subsystem is to load models and ports. After registering atomic models by adding pointers to their constructors into a model admin table (a hash table that serves as an atomic model object database), and top ports by adding them into a port admin table (a hash table that serves as a port object database), the Main Runtime subsystem constructs the DEVS model hierarchy. This is done by parsing the model file that contains the components and their relations (i.e., Atomic and Coupled models, their links/couplings – EIC, IC, EOC – and ports) and calling the Model Admin and the Processor Admin to construct a model hierarchy tree and a simulator/coordinator tree. These two trees are constructed in parallel, i.e., when the model admin adds a node; the processor admin also adds a corresponding execution node providing a one-to-one relationship. The model hierarchy tree belongs to the model class and has atomic (leaf nodes) and coupled models (non-leaf nodes) as its node while the processor hierarchy tree has simulators (leaf nodes) and coordinators (non-leaf nodes) as its nodes. As a Flattened Coordinator is used, the coupled models are eliminated from the model hierarchy tree, and all the atomic model port links are rewired to bypass the coupled models.

Once the models are loaded, the control is passed to the Root Coordinator that manages the rest of the execution by monitoring signals from the environment, handling scheduling, and passing messages.

The hardware component interface allows direct access to the microcontroller, and the entire application holds in the onboard memory. We were able to interface with STMicroelectronics' STM32 peripheral libraries, and this can be done with other vendors as well. Model continuity is achieved, as the DEVS model is not changed throughout development. This design is also portable as models developed can be run on bare metal by specifying the necessary port drivers to interact with real hardware components. As mentioned, the implementation of the port/driver concepts greatly increases this portability through the encapsulation and generalization of I/O devices allowing for simple addition of new devices.

The functionality of some functions run by the OS was reproduced through the creation of functions with the same signature but with a re-designed implementation that takes into account the limitations and environment of the target platform. For instance, Xenomai provides real-time guarantees through the implementation of constrained functions as well as the use of a real-time scheduler. In bare metal, timing is now controlled at the clock level through periodic timer interrupts to manage scheduling, and at the model level through model specification and model-checking of the timing constraints. A timer is set to trigger at every 1 μs. In addition, the removal of Xenomai also required changing the Root Coordinator, which would sleep until the next internal transition is scheduled, periodically verifying that an external event has not occurred. If an external event occurred, the event would 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 would place the microcontroller into low power mode, and it would wait for the next external event.

MCU device libraries allow access and configuration of hardware components. We integrated peripheral libraries in the hardware-dependent kernel. When defining input and output ports connected to hardware, the user can reuse methods provided by the hardware abstract library to get value from sensors or actuate on external devices. This includes ready-to-use methods to include commonly used hardware components (e.g., ultrasonic, light sensors).

The only file referenced by E-CD++ at runtime is the model file. In fact, models are loaded at runtime reading and interpretating the model file. Since we do not have an OS with a file system, we developed a pseudo file system to maintain continuity between desktop model simulation and the target model real-time execution. To mimic this behavior, 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 table that 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, file size.

Early integration of the bare-metal E-CD++ was done on an MCBSTM32F200 evaluation board. Developed by Keil, the board includes the STM32F207IG ARM Cortex-M3 based microcontroller, which has a clock speed of 120 MHz, 1 MB of ROM, and 128 KB of RAM. Through the implementation of drivers for the different I/O devices on the evaluation board, we performed early integration testing which demonstrated the feasibility of the bare-metal implementation. On the software platform side, Eclipse was used along with the GNU ARM bare metal toolchain 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 the development. This design is also portable, as the software core of E-CD++ has not changed (all that has changed are 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 the simple addition of new devices.

4. Building a Digital Quadruplet: Urban Futures Transportation Charrette

In this section we describe the context of a project for which we have used the bare-metal E-CD++ to build an autonomous vehicle integrated into a Digital Quadruplet. We focus on the vehicle integration for the case study, the definition and implementation of the embedded application, and we show how the modeling method can be used to build these models (which have been developed by Engineering students without previous expertise in DEVS or BIM and who were able to build the Digital Quadruplet with minimal support).

The Digital Quadruple was developed under the umbrella of the Urban Futures research project ((http://urbanfutures.ca/designcharrette-transport). The objective of Urban Futures is to conduct multidisciplinary research in the area of digital technologies for urbanism. To do so, experts from industry, government, academia, and not-for-profit organizations joined to question and challenge operational dimensions of ‘‘smart city’’ technologies. This is an important topic in the Canadian context. In 2016, Statistics Canada reported that over 80% of Canadians are living in urban areas and that the trend is accelerating. The calibration, coordination, and communication among intelligent systems needs problem-solving as an iterative, collaborative, and inclusive design exercise. The project activities were organized as a series of intensive design charrettes, in which participants discussed, debated, and developed ideas in reaction to themed ‘‘what-if’’ scenarios. The case study presented here focuses on the Transportation charrette for Carleton University, in Ottawa, ON, Canada. In this case study, depicted in Figure 7, we consider the deployment of autonomous vehicles (bottom right of the figure) that circulate through the campus, transporting individuals and goods from one building to the next, and stopping in parking lots, public transportation stations (light rail, bus), and specific stops throughout the campus.

Figure 7. Urban Futures: Transport Charrette (http://urbanfutures.ca/designcharrette-transport).

As we can see in Figure 8(a) and (b), we started by building a Digital Twin of Carleton University campus. A more detailed figure of this first step can be seen in Figure 9. We established a process using Autodesk Revit and Autodesk Dynamo to extract information of the whole campus to be used for simulation. This includes a cloud point with buildings, roads, transportation data, traffic signs, and sources of pedestrians (buildings, public transportation). This information is given as initialization data for DEVS models in CD++. We built a simulation model of the autonomous vehicle using CD++, which can navigate the digital campus in simulated fashion (shown in Figure 8(c)). We used the same tools for visualization analysis, to study the behavior of the system in the Digital Twin. The integration of the DEVS models and the Digital Twin conform to the campus Digital Triplet.

In the following sections we describe the models used to build the physical system at scale (Figure 8(d)), based

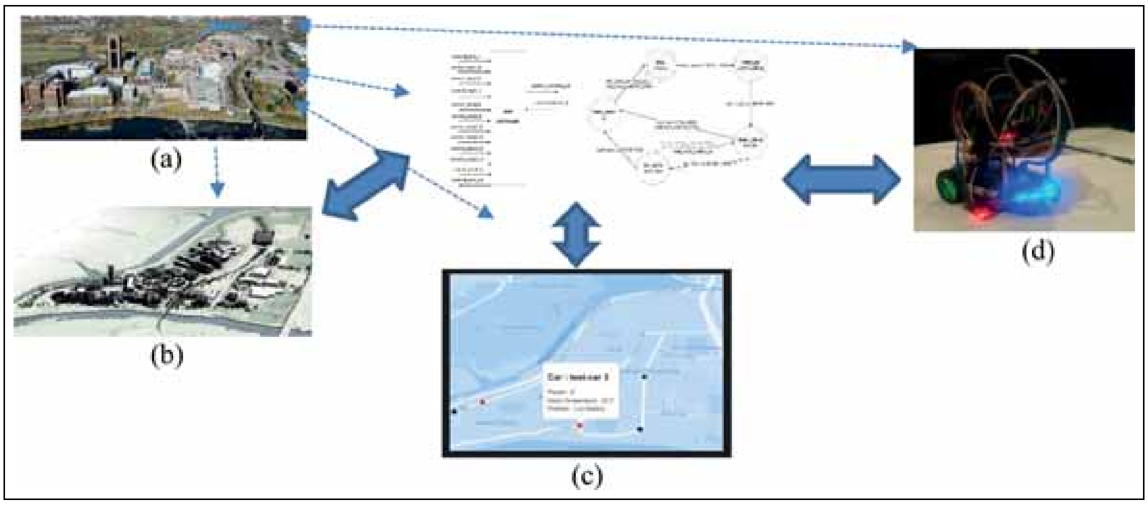

Figure 8. (a) Real System; (b) Digital Twin; (c) Digital Triplet; (d) Digital Quadruplet.

on models of the real system and a hardware implementation integrated to the Digital Quadruplet, which can be used for experimentation mixing simulation, visualization, and real-time control. The objective is to show how E-CD++ can be used for modeling, simulation, and real-time execution of the models.

4.1. The model

The first step in our methodology, as discussed earlier, is to build a model of the system of interest using DEVS. Figure 10 illustrates the resulting DEVS model hierarchy for the case study. A Central Computer contains the BIM models, and it is used to conduct visualization of the running models (which was shown in Figure 8(c)). The three main components are in charge of geolocating the vehicle (using RFID tags that are activated when the vehicle arrives in a given station), converting the information from the RFID into geolocation (which is sent to the BIM model for display), and driving the vehicle, done by the SeedBotDriver.

All the components are defined as a DEVS model. For instance, the SeedBotDriver, shown in the bottom part of the figure, is defined as a coupled model with four input ports to give/receive information about (start/stop external commands, distance and direction sensors values), and two output ports to send commands to vehicle motors. The vehicle top model is made of three coupled models: the *Sensor Unit*, *Control Unit*, and *Movement Unit*. The *Sensor Unit* contains two *Direction Sensor* models (left and right), and a distance sensor, defined using three atomic models. The distance sensor calculates the distance to an object and transmits the readings to the *Control Unit* for processing. The direction sensors send signals to the sensor unit indicating if an object was detected or not in that direction. The *Control Unit* is built as a composite of two atomic models: the *Sensor Controller* and the *Movement Controller*. The *Sensor Controller* receives the distance and direction readings and sends information to the Movement Controller to specify whether the vehicle is free to move, or has encountered an obstacle (in front or sideways).

The *Movement Controller* receives obstacle signals from the Sensor Controller and sends appropriate commands to the Movement Unit (which contains controllers for left and right motors). According to the sensor information, the *Movement Unit* sends ‘‘go forward’’ commands to both motors (to continue straight), or only to the right or left motors, in forward/reverse direction, to avoid obstacles depending on the obstacle location (front, left, or right). Each of these models is defined using DEVS; we show the definition of the *Sensor Controller* model as an example.

Figure 11 illustrates the state transitions using a DEVS Graph representing the Sensor Controller’s behavior. The diagram summarizes the behavior of the DEVS atomic component by presenting the states, transitions, inputs, outputs, and state durations graphically.[38] The circles represent states (the double circle is the initial state), which have a name and a duration shown in the circle. The continuous edges between the states represent the external transitions, which include the names of the input ports, the input value, and any condition on the input (with format ‘‘port?value’’). The dotted lines represent the internal transitions and the associated outputs (with format ‘‘port!value’’).

The *Sensor Controller* starts in the IDLE state and remains in that state until a *start* command is received. In that case, the external transition function executes, and the *Sensor Controller* state changes to PREP_RX. At this stage, it waits for the predefined time ta=scRxPrepTime

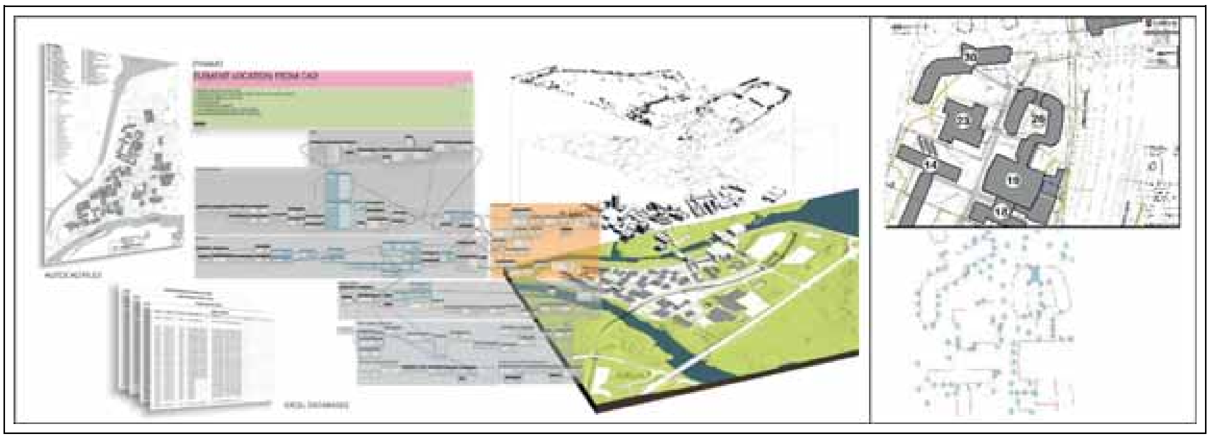

Figure 9. Carleton University Digital Twin (https://sustain.sce.carleton.ca).

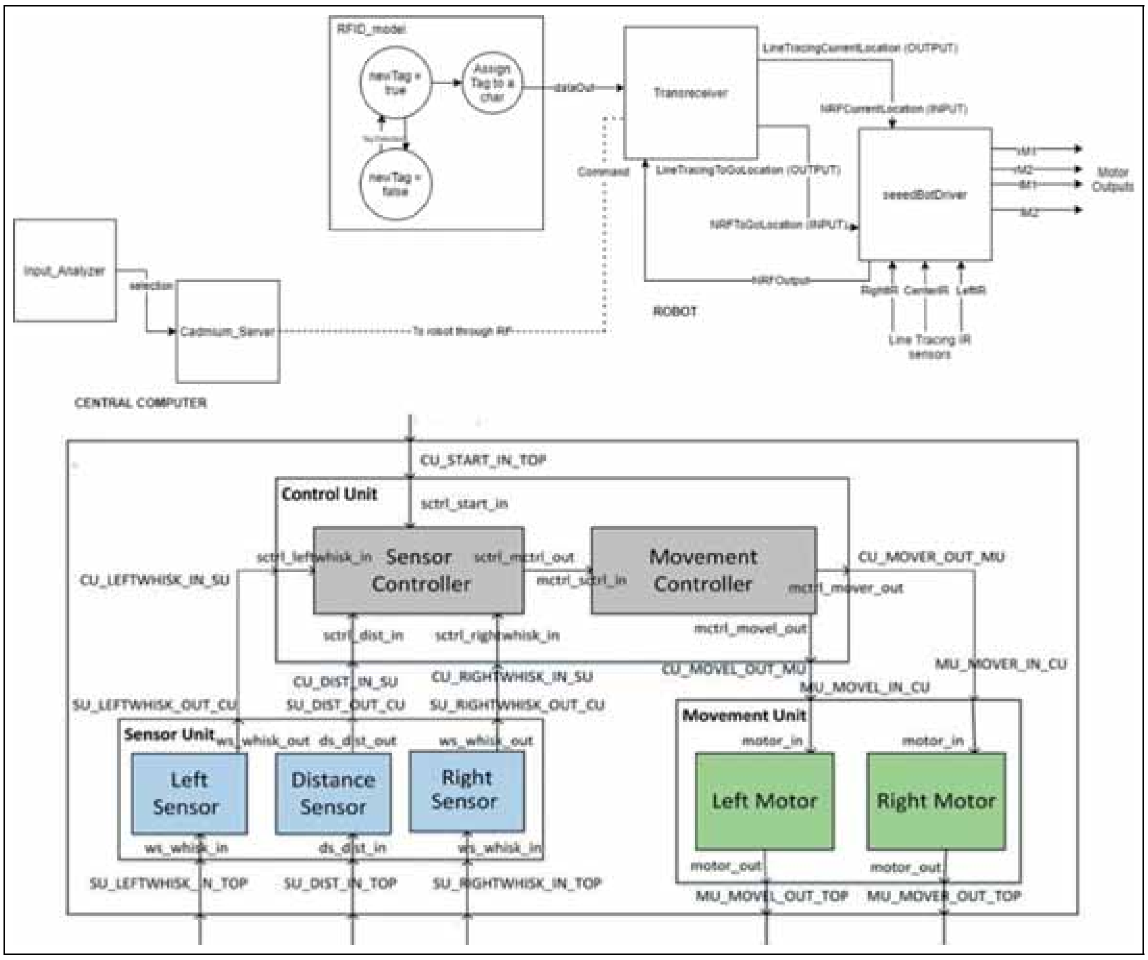

Figure 10. Vehicle model DEVS hierarchy.

after which a default NO_OBSTACLE signal is sent to the Movement Controller and an internal transition is triggered changing its state to WAIT_DATA. *Sensor Controller* waits in this state until it receives a signal from one of the sensors. When a signal is received, if it indicates to *stop*, the external transition causes *Sensor Controller* to go to

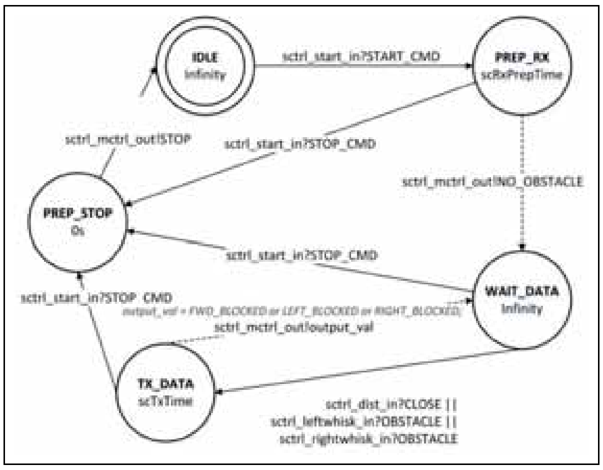

Figure 11. Sensor Controller DEVS graph.

the PREP_STOP state, at which it will immediately send a stop signal and then transition back to the IDLE state. However, if the received signal is different, *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 indicating whether the vehicle has encountered an object in front or a side sensor detected with an obstacle, and transitions back to the WAIT_DATA state. At any point in time, if *Sensor Controller* receives a manual stop signal (STOP_CMD), it will execute an external transition to the PREP_STOP state to stop all activities.

Using the model transformation defined in Saadawi and Wainer,[35] each of the models in the system can be converted into a Timed Automata and verified formally using model-checking (for instance, using a tool like UPPAAL). As stated earlier, this section will only focus on showing how the models are defined and then executed in the context of simulation or real-time execution (the reader interested in formal verification of the model can refer to Hwang[34]).

4.2. *Model definition in E-CD++*

E-CD++ provides a mechanism to program DEVS models' hierarchical structures. The coupled model definitions are defined using a built-in language which uses the formal definition of the DEVS coupled models. The following code snippet describes the *Sensor* and *Movement Controllers* specification components of the *Control Unit*, in accordance with the model diagram described in section 4.1.

The Model file snippet (Figure 12) 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, CU_LEFT_IN_SU,CU_RIGHT_IN_SU and CU_DISTANCE_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 atomic model presented earlier in Figure 11 describes the transition and output functions of the *Sensor Controller*. A section of the model in CD++ is presented in Figure 13.

The code snippet first shows a portion of the external transition function that describes the transition from state WAIT_DATA to either TX_DATA depending on the value (sensor_input) of the incoming signal from the *Distance Sensor* received on port *sctrl_dist_in* (as shown

```
[ControlUnit]
components : SCtrl@SensorController MCtrl@MovementController

in  : CU_START_IN_TOP CU_LEFT_IN_SU CU_RIGHT_IN_SU CU_DISTANCE_IN_SU
out : CU_MOVER_OUT_MU CU_MOVEL_OUT_MU

%input connections
Link : CU_START_IN_TOP sctrl_start_in@SCtrl
Link : CU_LEFT_IN_SU sctrl_left_in@SCtrl
Link : CU_RIGHT_IN_SU sctrl_right_in@SCtrl
Link : CU_DISTANCE_IN_SU sctrl_distance_in@SCtrl

%output connections
Link : mctrl_moveR_out@MCtrl CU_MOVER_OUT_MU
Link : mctrl_moveL_out@MCtrl CU_MOVEL_OUT_MU

%internal connections
Link : sctrl_mctrl_out@SCtrl mctrl_sctrl_in@MCtrl
```

Figure 12. Model file excerpt.

```
1  Model &SensorController::externalFunction( const ExternalMessage &msg ) {
2  ...
3  if (msg.port() == sctrl_dist_in)         // Distance sensor signal received
4   if(state == WAIT_DATA) {               // Sensor controller was waiting for data
5       if(msg.value()==CLOSE) {              // Destination Reached
6          sensor_input = FRONT_CLOSE;          // Set sensor_input value
7          state=TX_DATA;                       // change state to TX_DATA
8           holdIn(Atomic::active, scTxTime );  // transmit data after scTxTime
9      }
10 }
11
12 Model &SensorController::internalFunction( const InternalMessage & ) {
13    switch (state){
14          ...
15       case PREP_RX:            // Ready to receive data from sensors
16       case TX_DATA:            // Just transmitted data to movement controller
17          state = WAIT_DATA;// Wait for sensor data
18          passivate();      // stay in this state until a signal is received
19          break;
20    }
21 }
22
23 Model &SensorController::outputFunction( const InternalMessage &msg ) {
24    switch (state){
25    ...
26       case TX_DATA: {            // Sensor Controller is in transmitting data state
27       int output_val;
28
29       if(sensor_input==FRONT_CLOSE)              output_val = FWD_BLOCKED;
30          else if(sensor_input==LEFT_OBSTACLE)    output_val = LEFT_BLOCKED;
31          else if(sensor_input==RIGHT_OBSTACLE)   output_val = RIGHT_BLOCKED;
32          else                                    output_val = NO_OBSTACLE;
33        sendOutput(msg.time(),sctrl_mctrl_out, output_val); // Output sent to MCtrl
34        break;
35       }
36    }
37 }
```

Figure 13. Sensor Controller code snippet.

in Figure 11). Lines 12 to 20 show a portion of the internal transition function describing the transition from TX_DATA to WAIT_DATA. Finally, lines 23 to 37 show a portion of the output function's behavior at state TX_DATA. The output function sets the output signal (FWD_BLOCKED, LEFT_BLOCKED, RIGHT_BLOCKED or NO_OBSTACLE) to send to the *Movement Controller* through port sctrl_mctrl_out.

Using these model classes and the core components of E-CD++, different scenarios were tested early on the development platform using simulation mode. Once the verification step is completed for each of the components, they can be transferred incrementally to the target platform. In order to do this, each driver is associated with specific commands related to the hardware component it interacts with and the corresponding input/output ports. In this case, we deployed these models in different ARM boards. In Figure 14 we show one configuration built on a Parallax shield. The native code is downloaded directly in memory via ST-LINK, an in-circuit programmer for the STM32 microcontroller families. This interface module is enabled with JTAG/serial wire debugging 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 interface and the comprehensive standard peripheral libraries offered by STMicroelectronics in this case. These two elements can be seamlessly integrated, compiled to the native bytecode, and result in a DEVS-based firmware able to control the peripherals and respond to diverse external stimuli.

Different tests conducted progressively on each of the components allowed us to show the idea of model continuity. The models defined formally as in Figure 11 were thoroughly studied through E-CD++ simulation interface. Each of the simulated components were replaced by the corresponding hardware components and their controllers in the target platforms. As discussed in the paper, the

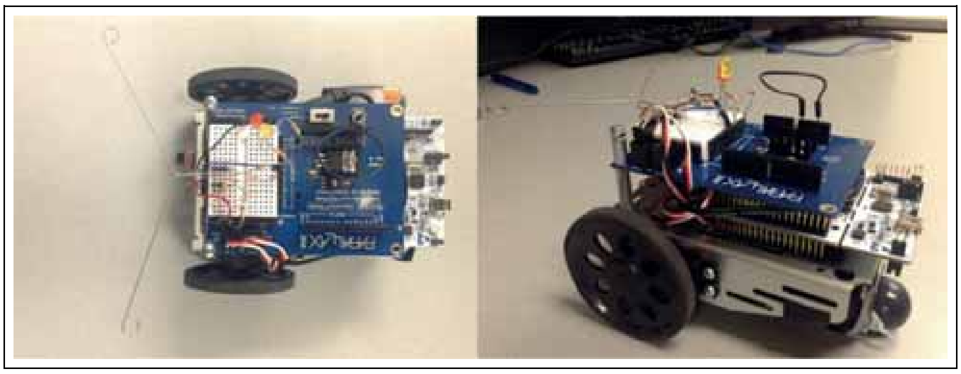
Figure 14. Parallax Shield with Nucleo Board.

Table 1. Port mapping.

Port name	Port value	Hardware command	Comment/description
START_IN	10	START_CMD	Start command
	11	STOP_CMD	Stop command
DISTANCE_IN	x	x > 10?FAR:CLOSE	Distance in cm. If the distance is less or equal to 10 cm, the object is close
LEFT_IN	0	NO_OBSTACLE	No obstacle detected
	1	OBSTACLE	Object detected
RIGHT_IN	0	NO_OBSTACLE	No object detected
	1	OBSTACLE	Object detected
MOVER_OUT/	0	STOP	Stops the motor
MOVEL_OUT	1	FORWARD	Spins clockwise
	2	REVERSE	Spins anticlockwise

models in both simulated and real-time control mode are exactly the same and the bare-hardware kernel can understand the DEVS models and execute them in real-time mode, directly on the hardware platform without any intermediate software. The final deployment was made on the final target and models run on the microcontroller. Table 1 shows the port mapping table for some of the tests conducted.

An example of simulated events injected into the system is as follows. The system starts by sending a START_CMD (10) input through the START_IN input port. Then, at 2'01'', a value of 30 (meaning that an object is at 30 cm in front of the vehicle) is sent through the DISTANCE_IN input port. To illustrate the situations when the vehicle gets close to the object, a value of 10 is sent through the DISTANCE_IN port at 07'50''. Different values are then sent through the directions port to test how the system behaves after a direction detects an object (e.g., right direction at 13' and 18', left direction at 17' and 18'05''). Afterward, a short distance, 5 cm, is sent through the distance port at 19'. The remaining input events should not trigger any change in movement since they do not indicate the presence of an obstacle.

Table 2 shows and describes the results that we obtained.

The resulting behavior is similar to the one defined in the controller models. Indeed, when the vehicle detects a close object, it stops and turns in order to avoid the obstacle, and stops again before moving forward. Similarly when an obstacle is detected, the vehicle avoids the obstacle, as expected. The same inputs were used on the ARM board. In the next figure, some of the inputs are shown as well as their corresponding outputs (in bold). Microseconds are shown in the logs since we used a 32-bit timer that allows such precision.

```
00:00:00:000:030 START_IN 10
00:00:00:150:002 mover_out 1
00:00:00:150:002 movel_out 1
00:02:01:000:031 DISTANCE_IN 30
00:05:07:000:036 DISTANCE_IN 20
00:07:50:000:022 DISTANCE_IN 10
00:07:50:060:200 mover_out 0
00:07:50:060:200 movel_out 0
00:07:50:160:200 mover_out 2
...
```

Table 2. Results of Line Tracking Vehicle's model in E-CD++.

Input	Output	Description
1. 00:00:00:000 START_IN START_CMD	00:00:00:150 mover_out 1 00:00:00:150 movel_out 1	System START. The vehicle moves forward
2. 00:02:01:000 DISTANCE_IN 30cm	-	Ignored – no obstacle
3. 00:05:07:000 DISTANCE_IN 20cm	-	Ignored – no obstacle
4. 00:07:50:700 DISTANCE_IN 10cm	00:07:50:060 mover_out 0 00:07:50:060 movel_out 0	STOP
	00:07:50:160 mover_out 2 **00:07:50:160 movel_out 1**	**TURN RIGHT**
	00:07:51:110 mover_out 0 00:07:51:110 movel_out 0	STOP
	00:07:51:210 mover_out 1 **00:07:51:210 movel_out 1**	**FWD**
5. 00:08:05:000 LEFT_IN NO_OBSTACLE	-	Ignored – no obstacle
6. 00:09:10:000 RIGHT_IN NO_OBSTACLE	-	Ignored – no obstacle
7. 00:10:10:000 DISTANCE_IN 20cm	-	Ignored – no obstacle
8. 00:13:00:000 RIGHT_IN OBSTACLE	00:13:00:050 mover_out 0 00:13:00:050 movel_out 0	STOP
	00:13:00:150 mover_out 1 **00:13:00:150 movel_out 2**	**TURN LEFT**
	00:13:01:100 mover_out 0 00:13:01:100 movel_out 0	STOP
	00:13:01:200 mover_out 1 **00:13:01:200 movel_out 1**	**FWD**
9. 00:15:00:000 DISTANCE_IN 30cm	-	Ignored – no obstacle
10. 00:17:00:000 LEFT_IN OBSTACLE	00:17:00:050 mover_out 0 00:17:00:050 movel_out 0	STOP
	00:17:00:150 mover_out 2 **00:17:00:150 movel_out 1**	**TURN RIGHT**
	00:17:01:100 mover_out 0 00:17:01:100 movel_out 0	STOP
	00:17:01:200 mover_out 1 **00:17:01:200 movel_out 1**	**FWD**
11. 00:18:00:000 RIGHT_IN OBSTACLE	00:18:00:050 mover_out 0 00:18:00:050 movel_out 0	STOP
	00:18:00:150 mover_out 1 **00:18:00:150 movel_out 2**	**TURN LEFT**
	00:18:01:100 mover_out 0 00:18:01:100 movel_out 0	STOP
	00:18:01:200 mover_out 1 **00:18:01:200 movel_out 1**	**FWD**
12. 00:18:05:000 LEFT_IN OBSTACLE	00:18:05:050 mover_out 0 00:18:05:050 movel_out 0	STOP
	00:18:05:150 mover_out 2 **00:18:05:150 movel_out 1**	**TURN RIGHT**
	00:18:06:100 mover_out 0 00:18:06:100 movel_out 0	STOP
	00:18:06:200 mover_out 1 **00:18:06:200 movel_out 1**	**FWD**
13. 00:19:00:000 DISTANCE_IN 5cm	00:19:00:060 mover_out 0 00:19:00:060 movel_out 0	STOP
	00:19:00:160 mover_out 1 **00:19:00:160 movel_out 2**	**TURN LEFT**
	00:19:01:110 mover_out 0 00:19:01:110 movel_out 0	STOP
	00:19:01:210 mover_out 1 **00:19:01:210 movel_out 1**	**FWD**
14. 00:20:00:000 DISTANCE_IN 20cm	-	Ignored – no obstacle
15. 00:21:05:000 LEFT_IN NO_OBSTACLE	-	Ignored – no obstacle
16. 00:21:15:000 RIGHT_IN NO_OBSTACLE		Ignored – no obstacle

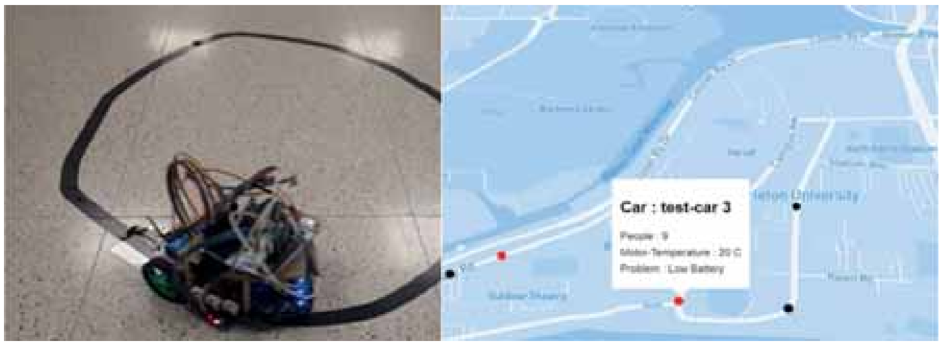

Figure 15. Digital Quadruplet experimentation.

```
00:13:00:150:214 movel_out 2
00:13:01:100:214 mover_out 0
00:13:01:100:214 movel_out 0
```

The model running in E-CD++ stand-alone software reproduces the simulated results without any modifications to the original models. After running various scenarios to verify the model behavior on the board, the driver interfaces were mapped with the vehicle sensors and actuators. The DISTANCE_IN driver is associated with a digital distance sensor for measuring the distance to nearby objects and LEFT_IN and RIGHT_IN are linked to the vehicle directions. The output drivers MOVER_OUT and MOVEL_OUT are connected to the two servomotors.

The integration of the different components of the Digital Quadruplets was completed. Figure 15 shows the integration of the BIM model representing Carleton campus, various simulated vehicles executed, and the physical model at scale integrated with the rest of the components. We can execute various tests, for example we show the case where the vehicle is running out of battery and reports the results remotely, and visualization results are put into the Digital Twin to report the system operator. Various videos showing the execution on the target platforms are available at https://bit.ly/2FMZS5P.

5. Conclusion

We introduced the concept of Digital Quadruplet: a 3D virtual representation of the physical world under study, a Discrete-Event formal model of the system of interest which can be used for formal analysis as well as simulation studies, and a physical model of the real system under study for experimentation with the goal of improvement of the development of Embedded Real-Time Systems. We showed the Digital Quadruplet concept, and discussed how to use the Discrete-Event formal model as a center for both simulation and execution of the real-time embedded components with timing constraints. We introduced E-CD++, a DEVS kernel running on bare metal that runs on different hardware platforms and provides a DEVSRT-based execution engine that manages the execution while models behave like processes. We now have an OS independent platform that would be fully portable and loadable onto various development boards by removing its Linux dependency, which can be used to build the embedded RT components in digital quadruplets.

At present we are building a complete Digital Quadruplet of the 3rd Floor VSim building including controllers for HVAC, lighting, occupancy, fire alarms, integrating a complete BIM model of the floor, real-time data obtained by sensors installed in the building, and ventilation. The automated vehicle presented here will be integrated in the building to roam between the different labs exploring the use of UV light for disinfecting the laboratories under the current pandemic of coronavirus disease 2019 (COVID-19). The vehicle will be used to conduct different exploration analysis on possible configurations of the equipment and the cleaning schedule, automating this process without human intervention.

Acknowledgements

The authors thank Joseph Boi-Ukeme for an early version of this manuscript. Figure 1 includes results presented by Hoda Khalil. The example presented in Figure 7 is a part of the Urban Futures initiative by Prof. Steve Fai at CIMS, Carleton University (http://urbanfutures.ca/designcharrette-transport). Figure 9 was built by Nicolás Arellano and other CIMS members, as well as Professor Liam O’Brien (https://sustain.sce.carleton.ca). The experiment in Figure 15 shows results of a test case built by José Franco, Aryan Rashidi-Tabrizi, and Bruno St-Aubin.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research has been partially funded by NSERC and Autodesk.

ORCID iD

Gabriel Wainer https://orcid.org/0000-0003-3366-9184

References

1. Qing L and Yao C. *Real-time concepts for embedded systems*. Boca Raton: CRC Press, 2003.
2. Henzinger T and Sifakis J. The embedded systems design challenge. In: Jayadev M, Tobias N and Emil S (eds.) *14th International symposium on formal methods*. Berlin: Springer, 2006, pp. 1–15
3. Hu X and Zeigler B. Model Continuity in the Design of Dynamic Distributed Real-Time Systems. *IEEE Trans Syst Man Cybern A Syst Hum* 2005; 35(6): 867–878.
4. Edwards S, Lavagno L, Lee E, et al. Design of embedded systems: Formal models, validation, and synthesis. In: *Readings in hardware/software co-design*. Boca Raton, FL, 2001, p. 86.
5. Moallemi M and Wainer G. *Designing real-time systems using imprecise discrete-event system specifications*. Softw Pract Exp; in press.
6. Wainer G, Glinsky E and MacSween P. A model-driven technique for development of embedded systems based on the DEVS formalism. In: Sami B, Matthias B and Volker G (eds) *Model-driven software development*. Berlin: Springer, 2005, pp. 363–383.
7. Boschert S and Rosen R (2016) Digital twin—the simulation aspect. In: Hehenberger P and Bradley D (eds) *Mechatronic futures*. Cham: Springer. https://doi.org/10.1007/978-3-319-32156-1_5
8. Khajavi SH, Motlagh NH, Jaribion A, et al. Digital twin: vision, benefits, boundaries, and creation for buildings. *IEEE Access 2019*; 7: 147406–147419, 2019. doi: 10.1109/ACCESS.2019.2946515.
9. Gery E, Harel D and Palachi E. Rhapsody: a complete life-cycle model-based development system. In: Michael B, Luigia P and Kaisa S (eds) *Integrated formal methods*. Berlin: Springer, 2002, pp. 1–10.
10. Wainer G. *Applying modelling and simulation for development embedded systems. Proceedings of SummerSim 2019*, Berlin, Germany, 2019.
11. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation*. 2nd edn. San Diego, CA: Academic Press, 2000.
12. Marwedel P. *Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things*. 3rd edn. Cham: Springer, 2018.
13. Lee E and Sanjit A. *Introduction to embedded systems: A cyber-physical systems approach*. 3rd edn. Cambridge, MA: MIT Press, 2016.
14. Madlener F, Weingart J and Huss S. Verification of dynamically reconfigurable embedded systems by model transformation rules. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Scottsdale, Arizona, USA, October 24–29. ACM, 2010, pp. 33–40.
15. Aravantinos V, Voss S, Teufl S, et al. AutoFOCUS 3: tooling concepts for seamless, model-based development of embedded systems. *ACES-MB&WUCOR@ MoDELS* 2015; 19–26.
16. Henzinger T and Sifakis J. The discipline of embedded systems design. *Computer* 2007; 40(10): 32–40.
17. Ahmed A and Wolf W. Hardware/software interface codesign for embedded systems. *IEEE* 2005; 38(2): 63–69.
18. Tormo D, Vidal-Albalate R, Idkhajine L, et al. Study of system-on-chip devices to implement embedded real-time simulators of modular multi-level converters using high-level synthesis tools. *2018 IEEE International Conference on Industrial Technology (ICIT)*, Lyon, France, February 20–22, 2018, pp. 1447–1452.
19. Hu G, Ren S and Wang X. A comparison of C/C++-based Software/Hardware Co-design Description Languages. *The 9th International Conference for young computer scientists*, Hunan, China, November 19–21, 2008, pp. 1030–1034.
20. Riccobene B, Rosti A, Lavazza L, et al. SystemC/C-based model-driven design for embedded systems. *ACM Trans Embed Comput Syst (TECS)* 2009; 8(4): 30.
21. Bashir R, Lee S, Khan S, et al. UML models consistency management: Guidelines for software quality manager. *Int J Inform Manage* 2016; 36(6): 883–899.
22. Feiler P, Gluch D and Hudak J. *The architecture analysis & design language (AADL): An introduction (No. CMU/SEI-2006-TN-011)*. Carnegie-Mellon University, Pittsburgh PA Software Engineering Institute, 2006.
23. Murillo L, Mura M and Prevostini M. MDE support for HW/SW codesign: A UML-based design flow. In: Dominique B (ed.) *Advances in design methods from modeling languages for embedded systems and SoC's*. Dordrecht: Springer, 2010, pp. 19–37.
24. Nascimento F, Oliveira M and Wagner F. A model-driven engineering framework for embedded systems design. *Innov Syst Softw Eng* 2012: 8(1): 19–33.
25. MathWorks. The MathWorks website. *MathWorks Online*. Available at: http://www.mathworks.com/ (accessed 11 February 2020).
26. Balasubramanian K, Gokhale A, Karsai G, et al. Developing applications using model-driven design environments. *Computer* 2006; 39(2): 33–40.
27. Eker J, Janneck J, Lee E, et al. Taming heterogeneity: the Ptolemy approach. *Proc IEEE* 2003; 91(1): 127–144.
28. Basu A, Bozga M and Sifakis J. Modeling heterogeneous real-time components in BIP. *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, September 2006, pp. 3–12.
29. Vidal J, De Lamotte F, Gogniat G, et al. A co-design approach for embedded system modeling and code generation with UML and MARTE. *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, Nice, France, April 20–24, 2009, pp. 226–231.
30. Moreira T, Wehrmeister M, Pereira C, et al. Automatic code generation for embedded systems: From UML specifications to VHDL code. *Industrial Informatics (INDIN), 2010 8th IEEE International Conference*, Osaka, Japan, July 13–16, 2010, pp. 1085–1090.
31. Wainer G. *Discrete-event modeling and simulation: a practitioner's approach*. Boca Raton: CRC Press, 2009.

32. Furfaro A and Nigro L. A development methodology for embedded systems based on RT-DEVS. *Innov Syst Softw Eng* 2009; 5(2): 117–127.
33. Huang D and Sarjoughian H. Software and simulation modeling for real-time software-intensive systems. *Eight IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04). IEEE*, Budapest, Hungary, October 21–23, 2004, pp. 196–203.
34. Hwang M. Qualitative verification of finite and real-time DEVS networks. *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, Society for Computer Simulation International, Orlando, Florida, March 26–30, 2012, p. 43.
35. Saadawi H and Wainer G. Principles of DEVS Models Verification. *Simulation* 2013; 89(1): 41–67.
36. Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87(1–2): 113–132.
37. Gholami S and Sarjoughian H. Action-level real-time network-on-chip modeling. *Simul Model Pract Th* 2017; 77: 272–291.
38. Song H and Kim T. Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad crossing control example. *Simulation* 2005; 81(2): 119–136.
39. Moallemi M, Wainer G, Bergero F, et al. Component-oriented interoperation of real-time DEVS engines. *Proceedings of the 44th Annual Simulation Symposium of the Society for Computer Simulation International*, Boston, Massachusetts, April 3–7, 2011, pp. 127–134.
40. Moallemi M and Wainer G. A system-on-chip FPGA implementation of embedded CD++. *Proceedings of the 2009 Spring Simulation Multiconference*, Society for Computer Simulation International, San Diego, California, March 22–27, 2009, p. 153.
41. Castro R, Ramello I, Bonaventura M, et al. M&S-based design of embedded controllers on network processors. *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, Society for Computer Simulation International, Orlando, Florida, March 26–30, 2012, p. 32.
42. Dozio L and Mantegazza P. Real Time Distributed Control Systems Using RTAI. In: *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Hakodate, Hokkaido, Japan, May 14–16, 2003.
43. Wainer G and Glinsky E. Model-based development of embedded systems with RT-CD++. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Toronto, ON, 2004.
44. Moallemi M and Wainer G. 'Designing an interface for real-time and embedded DEVS,' *Proceedings of the 2010 Spring Simulation Multiconference*. Arlington, VA, 2010.
45. Wainer G, Glinsky E and Gutiérrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone Benchmark. *Simulation* 2011; 87(7): 555–580.

Author biographies

Daniella Niyonkuru is a Software Engineer for Production Infrastructure at Shopify where she builds a better, faster and more resilient platform. Before entering the SRE world, Daniella was an Aircraft System Software Specialist, and researched Formal Model Driven Development for Embedded Systems.

Gabriel Wainer is Professor in the Department of Systems and Computer Engineering at Carleton University. He is a member of the Board of Directors of SCS. He is the author of three books and over 350 research articles. Prof. Wainer is Special Issues Editor of SIMULATION, member of the Editorial Board of IEEE/AIP CISE, Wireless Networks (Elsevier), and others. He received the IBM Eclipse Innovation Award, the First Bernard P. Zeigler DEVS Modeling and Simulation Award, the SCS Outstanding Professional Award (2011), the SCS Distinguished Professional Award (2013), the SCS Distinguished Service Award (2015) and various Best Paper awards. He is a Fellow of SCS.