Bare-Metal Kernels for DEVS Model Execution in Embedded Systems

by

Daniella Niyonkuru, **B.Eng.**

A thesis submitted to the

Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

August, 2015

The undersigned hereby recommends to the

Faculty of Graduate and Postdoctoral Affairs

acceptance of the thesis

Bare-Metal Kernels for DEVS Model Execution in Embedded Systems

submitted by **Daniella Niyonkuru**, **B.Eng.**

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

---

Professor Gabriel Wainer, Thesis Supervisor

Professor Yvan Labiche, Chair,
Department of Systems and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

August, 2015

Abstract

With the rising popularity of embedded systems came new challenges due to ever-growing market application demands, increasing complexity and widening productivity gap. To deal with these issues, model-driven development promotes a higher level of abstraction during design and uses models as the primary artifacts that guide the product development. In fact, the fundamental principle is to construct a model of a system and then transform it into the real system.

We focus on DEMES, a model-driven development methodology based on the Discrete Event System Specification (DEVS) — that defines a formal Modeling and Simulation framework for discrete event dynamic systems —, and especially the transition from simulated platform to execution platform, i.e. the embedded hardware. In this dissertation, we present bare-metal real-time executives that allow DEVS models to be executed on a target platform without the need of an operating system. This is particularly important for target platforms with limited resources. In addition to the real-time executives, we introduce a hardware abstract layer that supports several hardware peripheral libraries and fosters fast prototyping. We also illustrate the DEMES-driven development cycle with a particular case study: a line tracking robot application. Our contributions have resulted in a reduced footprint, increased performance and enhanced platform portability.

To my brother, David, for always lighting me up.

To my mother, Pascasie, for always lifting me up.

And above all, to God, The Father Almighty:

Ephesians 3:20-21.

Acknowledgments

I thank Dr Gabriel Wainer whose invaluable advice and ongoing support helped through my graduate studies and research. One of my objectives in graduate school was to work with and learn from some of the best in the field, and I had this privilege under his supervision. The past two years have been both a challenging and rewarding experience.

I also thank the members of the Advanced Real-Time Simulation Laboratory for their help and comradeship.

Table of Contents

List of Tables

List of Figures

Nomenclature

AM	Atomic Model
API	Application Programming Interface
CM	Coupled Model
CMSIS	Cortex Microcontroller Software Interface Standard
DEMES	Discrete Event Methodology for Embedded Systems
DEVS	Discrete Event System Specification
E-CD++	Embedded CD++
E-CDBoost	Embedded CDBoost
ES	Embedded Systems
FSM	Finite State Machine
GGAD	General Graphical Advanced environment for DEVS
GPIO	General Purpose Input/Output
HAL	Hardware Abstract Layer
IDE	Integrated Development Environment
IoT	Internet of Things
M&S	Modeling and Simulation
MCU	Microcontroller
MDD	Model-Driven Development
OS	Operating System
PDEVS	Parallel DEVS
RC	Root Coordinator
RTES	Real-Time Embedded Systems
RTS	Real-Time Systems
RTOS	Real-Time Operating System
SPL	Standard Peripheral Libraries
TA	Timed Automata
UML	Universal Modeling Language
UML-RT	UML for Real-Time
VHDL	VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Overview

Over the last few decades, embedded systems have ascended to a position of prevalence in the world. They are ubiquitous, diverse, and present in various industries such as aerospace, consumer electronics, defense, medical equipment and transportation. Embedded systems are generally defined as computing systems with tightly coupled hardware and software designed for a specific purpose. Real-Time Embedded Systems (RTES) [1], in particular, are not only subject to functional and logical correctness; they must also produce results within strict timing constraints. Missing these deadlines may lead to significant loss and in some cases catastrophic consequences. For instance, an airplane's flight controller must respond correctly within the required deadline, or the result could be serious injuries or death. Besides, RTES usually operate in limited resource environments, are required to have a small memory footprint, be low-cost solutions and have low power consumption.

In addition to dealing with timeliness requirements, RTES design needs to deal

with hardware/software partition, and cope with target systems increasing scalability and complexity. However, there is a real shortage of effective design and implementation practices. Indeed, besides the inherent challenges of embedded systems, the ever-increasing demands for new applications and technological advances have caused system complexities to grow at an exponential rate. Therefore, traditional design methodologies that conquer hardware and software are becoming infeasible and making software the most costly and least reliable part of RTES [2], with deficiencies originating from two main weak areas: the development cycle and system verification. Model-based development offers a promising solution by raising the level of abstraction and promoting models to principal artifacts that drives development. Yet, most model-based techniques do not have a formal foundation and struggle with system verification.

Formal methods, on the other hand, facilitate system verification since they have a strict mathematical foundation. However, most formal methods are hard to scale up [3], and they usually do not consider the physical environment that the embedded system controls. A practical solution to the above problems is the use of formal Modeling and Simulation (M&S) that combines the advantages of a simulation based approach with the rigor of a formal methodology [4].

The Discrete Event Methodology for Embedded Systems (DEMES) [5] [6] is one such approach. It is based on the Discrete Event System Specification (DEVS) [7], a formalism that decomposes complex systems into basic (behavioral) models, and composite (structural) models. DEVS is especially suitable for RTES since it provides a rich structural representation of components, and formal means for explicit time specification, which is essential to RTES. The formal part of DEVS allows model-checking essential to verifying of the system properties and building provably correct

software; while the M&S aspect is useful to handle complex system designs, interact with the physical environment models, and perform extensive testing in risk-free environments. In addition, DEMES has various advantages when compared to existing methodologies, especially because most model-based methods are semiformal and do not provide direct model continuity. Model continuity, in particular, refers to preserving the model specification as much as possible through the development process. With DEMES, models are consistently used through the entire development cycle. They are formally defined in the design/specification phase, used to perform model-checking, run simulations, and incrementally replaced with hardware surrogates. The original models end up being deployed on the target hardware where they act as controllers.

One particular component essential to models deployment on the embedded platform is the real-time executive. This latter executes original models on the particular hardware. In this thesis, we present two DEVS-based kernels (composed of a real-time executive, a microkernel, and a hardware abstraction layer) that run original models directly on bare-metal. The objective is to be able to execute models directly on the target system hardware without the need of an Operating System. The new real-time execution engines provide functionalities similar to those of a real-time kernel, with formal models operating as system processes. The development of these DEVS-based real-time kernels involved the following tasks:

1. The development of stand-alone real-time executives able to run on bare-metal without the need of an intermediate operating system.

2. The design of a hardware abstraction layer that provides an interface between hardware specific components and general real-time executive components.

3. The integration of different tools (compiler, hardware debugger, embedded register viewer, etc.) into an IDE in order to effectively develop embedded bare-metal applications.

4. The development of applications on top of the designed real-time execution engines.

1.2 Contributions

In this thesis, we introduce two main contributions. The first is the development of DEVS-based real-time kernels that run on bare-metal. The second is the design of a hardware abstract layer that allows the execution engines to run on multiple microcontrollers.

The first contribution extends the applicability of M&S driven development by providing an OS independent DEVS execution engine. As a result, target devices such as low power microcontrollers where an operating system would require excessive resources, are now covered. Since RTES are pervasive and varied, we believe that solutions addressing current development shortcomings should be applicable to a wide range of devices in order to properly replace traditional techniques and be appealing to industry. We have designed DEVS-based kernels able to execute models on bare-metal on different ARM-based boards without the need of middleware RTOS.

Two real-time executives were developed: one that reuses the components of E-CD++ [4] [8], an existing DEVS real-time executive that required the services of a Linux real-time kernel, and another one built around a new Parallel DEVS (PDEVS) sequential architecture [9]. This latter improves the performance of the executive by using a different message-passing concept between model execution engines.

The second main contribution is the design of a hardware abstraction layer that

provides a clear separation between the execution engine components and hardware specific services. We have designed a wrapper around the popular ARM MBED library [10] and integrated it with DEVS components in order to provide rapid prototyping and ease the development task. This library also allows the user to build Internet of Things (IoT) applications since MBED provides such a platform.

Finally, to illustrate and test the new real-time executives, we have built different applications among which is a line tracking robot. We present its development cycle, show how to build its application on the E-CD++ "legacy" version, as well as the new real-time executive. We also extend it with IoT capabilities to demonstrate the connectivity possibilities offered by the wrapper library.

1.3 Publications

Three papers that describe our work have been either published or accepted for publication. Listed below, these papers present the design of the bare-metal execution engines, the design of the new simulator around a PDEVS sequential architecture, and the discrete event methodology for embedded systems.

- [11] Daniella Niyonkuru and Gabriel Wainer. "Towards a DEVS-based Operating System." In Proceedings of the 2015 ACM SIGSIM conference on Principles of advanced discrete simulation, London, UK, 2015.

- [9] Damian Vicino, Daniella Niyonkuru, Gabriel Wainer and Olivier Dalle. "Sequential PDEVS Architecture." Proceedings of the 2015 Spring Simulation Multiconference, Alexandria, VA, USA, 2015.

- [6] Daniella Niyonkuru and Gabriel Wainer. "Discrete Event Methodology for

Embedded Systems." Computing in Science & Engineering vol.17, no.5, pp.52-63, Sept.-Oct. 2015.

We are also currently working on the following journal articles:

- Daniella Niyonkuru and Gabriel Wainer. "A DEVS-Based Framework for Bare-Metal Embedded Applications." *ACM Transactions on Modeling and Simulation,* vol.(TBD): pp (TBD). (in progress)
- Daniella Niyonkuru and Gabriel Wainer. "Embedded CDBoost, Executing DEVS models in embedded systems." *Journal (TBD),* vol.(TBD): pp (TBD). (in progress)

1.4 Thesis Organization

The first three chapters of this document are introductory in nature. Chapter 3 describes the scope and purpose of the project, referring to the previous research outlined in chapter 2.

Chapter 4 and 5 describe how the problem stated in chapter 3 was tackled. Chapter 4 describes the real-time executives. We start by introducing the common bare-metal design in section 4.1. Section 4.2 is then related to the legacy version and outlines the software architecture and changes that were made to the existing design in order to remove existing Linux dependencies and allow bare-metal execution. Next, section 4.3 presents the real-time executive built around the new PDEVS Sequential Architecture. We explain the architecture and algorithms used to build this executive and the hardware integration process. After, chapter 5 presents the hardware related layers of our solution.

Chapter 6 studies extensively the development cycle of one of the applications we built: a line tracking robot. The system of interest, its model specification and

simulation results are respectively presented in section 6.1, 6.2 and 6.3. Section 6.4 and 6.5 show how to build the system application on top of the new executives, and also assess their performances in 6.6. We describe in 6.7 how to port easily the built application from one platform to another. Finally, in section 6.8 is an Internet of Things (IoT) application that illustrates how devices can be easily interconnected using our platform.

The remaining chapter (7) concludes the dissertation and discusses future work.

Chapter 2

Review of the State of the Art

The traditional design approaches — where systems are directly designed at the low hardware or software levels — that are still used in most embedded system designs may be passable for small and medium sized systems; however, they are quickly becoming infeasible due to the ever-increasing complexity and new application market demands. Technology advances allow the integration of a rising number of components on a single chip; and Moores law that states that the number of transistors on a chip doubles every 18 months, still holds. In contrast, design methods improve at a much slower rate. This has resulted in a problem known as the productivity gap. Traditional methods cannot reduce the productivity gap; therefore, alternative approaches that can improve the quality, correctness, and modularity of systems by advancing the analysis and verification of properties as early as possible in the design flow are needed. Model-based techniques are today the most promising solution [12] to lessen the productivity gap and enhance the quality, correctness, and modularity of software systems and subsystems. In the following section, we will first revisit traditional design techniques and briefly describe how model-based design ascended. Afterwards, we will focus on model-based development.

2.1 The Evolution of Embedded System Design

2.1.1 Traditional Design Methods

Since RTES are partly made of hardware and software, three important design aspects [13] have to be considered: the hardware design, the co-design of hardware and software, and the design of embedded software. Traditional design divides hardware and software design to conquer them separately. The design starts with an informal specification and then the decision is made on how functionality will be split between hardware and software [14]. 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 [13]. The implementation is obtained using classical top-down methodology (synthesis and compilation) [15]. Two types of methodologies subsequently 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 run-time 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]. Moreover, in this type of design, system architects may use C and C++ to describe the system at the system-level [16]. The C/C++ description is refined; and then translated into synthesizable HDL [17]. In a nutshell, traditional design methodologies trace their origins from either software or hardware traditions [2] but do not cover hardware-software co-design, and remain hard to verify against the initial specification. Consequently, long testing phases, error-prone products, and increased time to market are common. On the other hand, co-design [18] 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.

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 [19] in order to enable hardware/software co-design. Examples include SpecC, ImpulseC and SystemC [20]. The latter, for instance, combines synchronous hardware semantics with asynchronous execution mechanisms from C++. Although the previous co-design description languages were quite effective in system-level design, they were not enough.

Recent methodologies offer higher levels of abstraction and go beyond implementation platform independence. They are built on modeling languages such as the Unified Modeling Language (UML) [21] and the Architecture Analysis and Design Language (AADL) [22] [23]. These approaches are system architecture focused and model-centric, and therefore referred to as model-based techniques.

2.1.2 Model-Based Design

Model-Based Design (MBD) seeks to address heterogeneity, and targets system's increasing scalability and complexity early in the development cycle by using models to describe the system. Model-Based Engineering (MBE) [23] [24], more generally, refers to engineering practices where models are the central and indispensable artifacts throughout the products development cycle enclosing concept, development, deployment, operation and maintenance. MBE has emerged in a variety of guises including model-driven engineering(MDE) [25], model-driven development (MDD) [26], model-driven architecture (MDA) [27] [28] and model-centered development (MCD) [29]. In these processes, models drive the development process by going through a series of transformations - a more abstract model is refined into a less abstract one - until

they reach a final state where they are made executable either by code generation of model interpretation and ready to be deployed [30]. Model-driven development is therefore "simply the notion that we can construct a model of a system that we can then transform into the real thing" [31]. Examples of available commercial MBD tools are Matlab/Simulink [32], Rational Rhapsody [33], SCADE [34], Modelica [35] and NI LabView [36]. In the academic context, we can cite Ptolemy [37] and Metro II [38]. The previous modeling languages/tools have powerful capabilities and some have been successfully adopted in industry. However, most companies had to create or develop small domain-specific languages to compensate for the limitations of current techniques [39]. We will take a closer look at Matlab/Simulink and UML-based profiles to illustrate some of the existing shortcomings.

Simulink [32] is one of the most extensively used model-based toolchain. It is a block diagram environment for multidomain simulation and MBD. It supports simulation, automatic code generation, continuous test and verification of embedded systems [40]. Hierarchical subsystems are modeled with predefined library blocks. In particular, stateflow charts (discrete logic and model behavior definition) and simulink (for continuous dynamics) models are used to represent the system. To simulate the dynamic behavior of the modeled systems, Simulink provides fixed-step and variable-step ordinary differential equation solvers. Simulink models can also be configured to generate code in C/C++, HDL and PLC. The tool also supports model connection to hardware and hardware-in the-loop (HIL) simulation; Simulink provides built-in support for prototyping, testing, and running models on various target hardware(FPGA, low-cost embedded hardware such as Arduino and Lego Mindstorms). In addition, several toolboxes (e.g. Simulink Design Verifier, Simulink Verification and Validation, SystemTest, and Simulink Code Inspector) are available in Simulink for model checking, algorithm verification and validation according to certain industry standards (e.g.

DO-178D [41] and IEC 61508 [42]) [43].

Although the Mathworks development suite has powerful capabilities, its modeling languages lack formal and rigorous semantics for its models. Several initiatives such as [44], [45], [46] and [47] used different methods (communicating pushdown automata, tabular expressions) in an attempt to define formal semantics for the popular tool. However, only a subset of the modeling languages is covered, and models remain susceptible to interpretation errors that can be fatal for safety-critical embedded systems. Instead, SCADE (Safety Critical Application Development Environment) is sometimes preferred for safety-critical applications since it has formal pedigree (based on data flow graphs and finite state machine) and strong analysis capabilities. In addition to the absence of rigorous semantics, Simulink lacks a publicly accessible meta-model that would enable the integration with a larger metamodeling language such as UML [48].

UML, on the other hand, is already well established in software engineering and comes naturally as the de facto standard for MDD. A number of UML profiles have been proposed for modeling embedded systems, and include SysML (Systems Modeling Language) [49] and MARTE (Modeling and Analysis Real-Time and Embedded systems) [50]. SysML is for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities; while MARTE supports specification of real-time and embedded systems [51]. In addition to functional design, this profile adds constructs to describe the hardware and software (e.g. OS services) resources and defines specific properties to enable designers to perform timing and power consumption analysis. MARTE is very general and supported by several tools. However, due to the complexity it has to support, MARTE is still a work in progress [12] subject to evaluation and extension proposals [52] [53] [54]. Alternatively, SysML and MARTE tend to be combined

because there is complementarity of technology coverage [55] [56]. Indeed, SysML supports modeling of a whole spectrum of diverse engineering technologies (e.g. mechanical, electrical, and hydraulic), and MARTE focuses primarily on real-time and embedded systems software and supporting platforms. Plus, the two can complement each other in terms of the level of abstraction they cover: the broader scope of SysML makes it a natural choice for representing systems at a higher level, whereas MARTE is better suited for finer grained modeling (e.g. the software aspects) [57]. However, because the two profile definitions are not yet fully synchronized with each other, combining them may lead to syntactic and semantic conflicts [57] [58]. The other main drawback of UML-based MDD is the gap between models and their execution. This step was particularly hindered by the lack of precise semantics that makes UML models hard to verify and can lead to model implementation inconsistencies. This lead to the adoption of fUML (foundational UML) [59] - the first precise operational and base semantics for a subset of UML encompassing most object-oriented and activity modeling- and ALF (Action Language for Foundational UML) [60] - a textual action language designed to specify executable fUML behaviors. Now, efforts such as in [61] are being directed towards defining formal semantics for UML profiles such as MARTE and SysML since they remain specified in prose (at best) and their semantic definitions stay informal. This considerably limits interest of profiles and their practical usability in a context in which engineers look for rapid prototyping. Other researchers, such as [62] [63] and [64], focus on providing mechanisms to further bridge the gap between model and code, and alleviate the burden associated with learning an extra action language, i.e. ALF.

Another approach used to tam heterogeneity in RTES design is the use of different Models of Computation (MoC) to cover a wide spectrum of domain. It is specifically referred to as Model-Integrated Development [65] and based on the idea that one

language cannot be enough to cover all domains and regroups several existing methods instead. Ptolemy II is a framework based on this concept. It is a structured and hierarchical approach, uses specific MoC that defines how computation takes place among a structure of computational components and supports discrete event, process networks (PN), dataflow (SDF), synchronous/reactive (SR) and continuous models. In Ptolemy II, real-time systems are modeled using DE; however Ptolemy II DE models don't have a formal specification and need to be transformed into a formal specification to enable model-checking. In [66], Real-Time Maude is integrated with Ptolemy II to enable formal verification. This way of integrating formal methods with MDE is also referred to as formal model engineering. This latter combines the convenience of using an informal but intuitive modeling language with formal verification. Nonetheless, useful information might be lost in the process of transforming informal models into formal models. Another alternative to preserve consistency would be to use methods that are natively formal and model-based.

Although model-based approaches handle well 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 development cycle — to reduce the model-to-execution gap — and model specification semantics — to enable formal 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. In terms of system verification, formal methods are needed to help with system verification and prove the system correctness.

To meet the previous considerations, a formal methodology that provides model continuity can be applied. In the following sections, we will introduce DEMES [5], a formal model-based development methodology based on Discrete-EVent System specifications (DEVS) [67]. 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 offers a practical approach with a formal rigorous method in which models are consistently used throughout the development cycle.

2.2 Embedded System Design with DEVS

DEVS (Discrete EVent System Specification) is a Discrete Event Simulation formalism for modeling and simulating dynamic systems. The DEVS formalism decomposes complex system designs into basic (behavioral) models called atomic and composite (structural) models called coupled [67] (See Appendix A for details). It follows a precise rule set to define state changes of the modeled systems with regards to input events or time delay triggers. DEVS is particularly suitable for RTES as it provides a rich structural representation of components, and formal means for explicitly specifying their timing, which is central for real-time systems. It has been proven to be successful in different complex systems (e.g. [68], [69], [70], [71])

Besides, DEVS is the most general Discrete Event Formalism and many existing formalisms (e.g. Statecharts, Petri nets, Timed Automata ...) can be expressed as DEVS [5] (See figure 2.1 [72]) and allows existing system description translation(Verilog [73], VHDL [74] transformation to DEVS). Plus, DEVS theory does not only provide a rigorous methodology for model construction but also proposes an abstract simulation algorithm independent of the simulation mechanisms and the

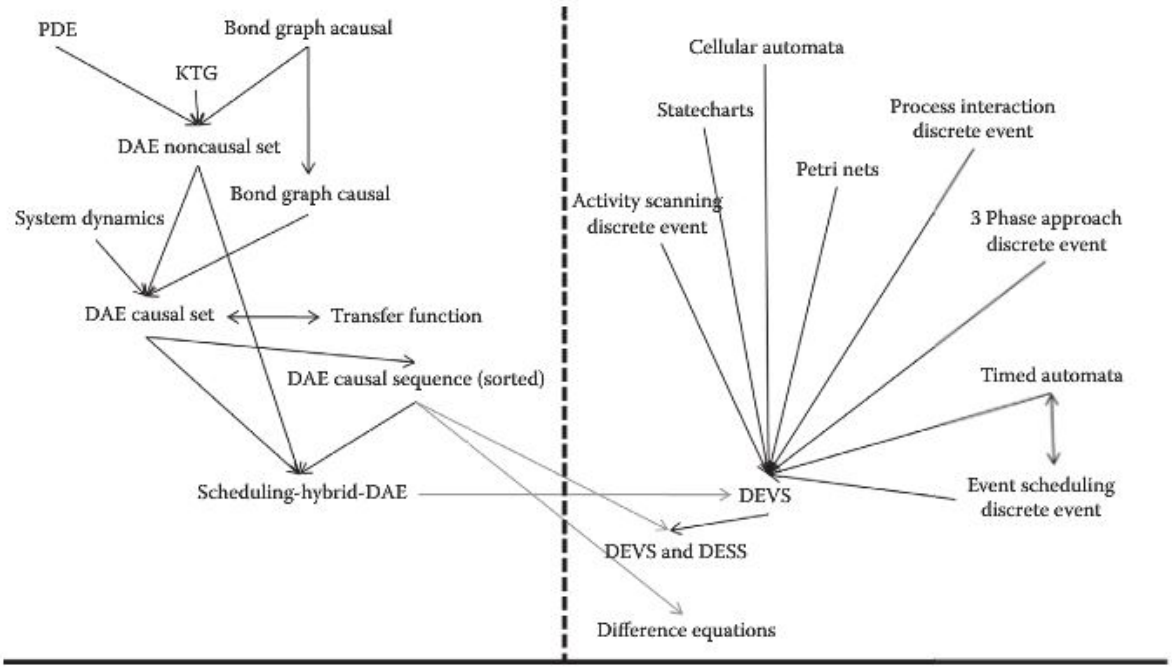

Figure 2.1: DEVS Transformation Graph

underlying hardware and middleware.

2.2.1 DEMES

DEMES focuses on bridging formal methods and M&S in order to analyze real-time systems and study their interaction with the physical environment while enabling original models to be part of the final product. This is achieved by using M&S for the initial stages, and replacing models incrementally with hardware surrogates and new software components without altering the original models. The transition can be done in incremental steps, incorporating models in the target environment after thorough testing in the simulated platform, allowing model reuse throughout the process.

Figure 2.2 shows the architecture of the DEMES methodology. A designer starts

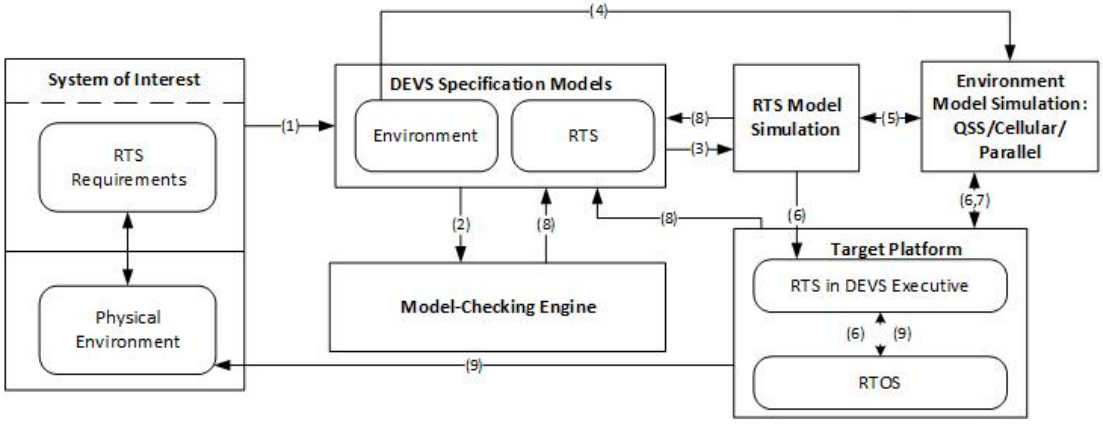

Figure 2.2: Discrete-Event Modeling of Embedded Systems (using DEVS)

(1) by modeling the System of Interest (a real time system and its environment) using formal specifications (DEVS or alternative techniques such as Statecharts, Modelica...). These models are converted into a DEVS representation, then transformed into Timed Automata, and verified using model-checking tools (2). In parallel with this formal verification phase (which can take a long time, in particular if state explosion happens during formal verification), the same models are used to test the components in a simulated DEVS environment (3). The physical environment can also be simulated (4) together with the RTS model under particular loads (5). Instead of obtaining general answers for all the possible cases (like those provided by model-checking), we can simulate individual behaviors of the different sub models under specific conditions. In brief, we can study system properties analytically, and complement the proofs using simulation, which can also be used for hardware/software co-design (and later, for training). These tested submodels can then be deployed incrementally into the target platform (6). A real-time executive executes the models on the particular hardware. If the hardware is not readily available, the software components can still be developed incrementally and tested against a model of the hardware to verify viability and take early design decisions. As the design process evolves, both

software and hardware models can be refined, progressively setting checkpoints in real prototypes. The executive allows to execute dynamic models and to schedule static and dynamic tasks. At this point, those parts that are still unverified in the formal and simulated environments are tested, increasing the confidence of the engineer into the implemented system. 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 in a spiral cycle (8), providing a consistent set of apparatus 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 existing methodologies. For instance, the methods discussed in section 2.1. were at most semi-formal and do not provide direct model continuity. In [75], 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 methods like DEVS provide sound syntax/semantics for structure, behavior, time representation and composition, which lend themselves to well-defined computation [76] [4]. Its expressiveness also allows to span multiple domain while remaining publicly accessible for integration with common standard modeling language such as UML [77] and some of its popular profiles [78]. DEVS has also been used in the past to bridge the gap between UML models and their execution such as in [79] and [80].

Moreover, the DEMES approach offers the following advantages [5] [81]:

- Reliability: logical and timing correctness rely on DEVS system theoretical roots and sound mathematical theory;

- Model reuse: DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition, 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.

2.2.2 Model-Checking with DEMES

Model-checking allows the designer to verify models correctness and eventually produce formally correct software. Therefore, deployed systems will have a very high reliability as the formal verification permits error detection at the early stages of the design. When models used for M&S are formal, their correctness is verifiable. Another advantage of executable models is that they can be deployed to the target platform, thus providing the opportunity to use the controller model not only for simulations but also as the actual code executing on the target hardware. Hence, the verified model is itself the final implementation executing in real time. This prevents any new errors that might appear during transformation of the verified models into an implementation, therefore guaranteeing a high degree of correctness and reliability.

To verify DEVS models, we use a class of rational time-advance DEVS called RTA-DEVS [82] and then transform RTA-DEVS models into equivalent Timed Automata (TA) that are then used to formally verify the desired properties using the UPPAAL [83] [84] model checker tool. RTA-DEVS was introduced to provide the modeler with a formalism that is expressive and sufficient to model complex systems behavior while being verifiable through formal model-checking techniques. RTA-DEVS is a subclass of DEVS that restricts the time advance function to nonnegative rational numbers and the elapsed time in a state used in the external transition to be a nonnegative rational number. These restrictions mean that we will have nonnegative rational constants in guards of the resulting TA model and ensure termination of the reachability analysis algorithms implemented in UPPAAL.

[85] presents a case study using a controller for an e-puck robotic application and showed the practicality of our approach.

Apart from formal verification, the other essential aspect that new approaches ought to offer is model execution on the hardware target. From this point on, we will focus on this feature and explore existing solutions.

2.2.3 Applicability to Low Level Applications

Recent research has focused on DEVS application to low-level applications commonly found in embedded systems consisting of computer hardware and real-time software. Existing DEVS based development environments for RTES include, for instance, DEVSJAVA [69], a Java DEVS-based simulator that supports high-level modeling; RTDEVS/CORBA [70] [86], a DEVS implementation based on real time CORBA communication middleware; and PowerDEVS [87] a tool for hybrid system modeling and real time simulation. In [76], the authors show how model continuity can be used in the design of dynamic distributed real-time systems. [88] presents

an application of the DEVS framework to the design and safety analysis of a RTES (a railroad crossing control system). In [71], an embedded control system model is built and its exhaustive verification temporal analysis done using Uppaal timed automata. In [89], model reuse and interoperability were shown by interfacing ECD++ and PowerDEVS. A System-On-Chip FPGA implementation of Embedded CD++ was presented in [90] and a M&S-based design of embedded controllers on network processors in [91]. The platform limitations remain significant compared to the traditional methods: In [76], [69], [70], [81] and [92] where implementation requires Java, the target hardware should be able to support the Java-implemented DEVS real-time execution environment. In [93] the authors presented a DEVS based real-time system on a TINI chip which has limited memory and processing ability. However, this requires Java Virtual Memory and Java class libraries availability on the chip. In [87], Linux RTAI kernel is required for PowerDEVS. The Embedded CD++ (E-CD++) developed by our team [8] relied on a variant of the Linux kernel. In [91], E-CD++ was embedded on the Core processor of an Intel IXP2400 Network Processor that runs RT Linux. Hence, it ran in the Linux User Space and requires Linux Kernel services. In [90], 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 [94]. This latter provided hard real-time functionality to the Linux kernel. In the next section, E-CD++ software components will be presented and its implementation explained.

2.3 Embedded CD++, a DEVS-based Tool

The DEVS formalism proposes a framework for model construction and defines an abstract simulation mechanism that is independent of the model itself. This mechanism provides a high-level implementation detail for the DEVS framework, and can be feasibly implemented by computer software. E-CD++ [8] is a real time implementation, based on the CD++ simulator [95] (a DEVS-based framework), and RT-CD++ [96] (an extension of CD++ for real-time simulation). E-CD++ supports modeling real-time systems by converting the CD++ virtual time-advance function to real-time, and provides an RT simulation platform for verification of such models. Figure 2.3 [4] illustrates the E-CD++ development framework. The embedded

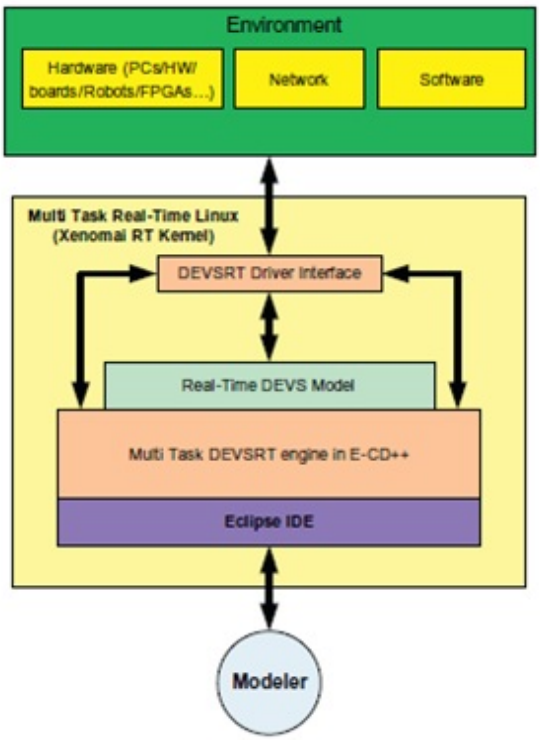

Figure 2.3: E-CD++ Layers

platform with the external environment is shown in this layered approach representing the cross-platform development of models. The modeler defines models using a high-level DEVS language combined with C++ code if needed, which provides the application layer. These real-time models are then interpreted and executed by the DEVSRT (DEVS in Real-Time) engine [97].

2.3.1 Features and Software Components Overview

To allow for direct replacement of models with external entities, the I/O ports of E-CD++ models implement the formal interfacing mechanism of DEVSRT in the Driver Interface layer. The underlying middleware is a real-time kernel and the runtime objects are imported to this platform as RT tasks. The E-CD++ execution engine uses the Xenomai real-time kernel [94] with multi-tasking services to implement DEVSRT. The user models and the driver objects were merged with the E-CD++ core objects; and the entire combination was compiled to produce an executable.

E-CD++ also include several features. The Eclipse IDE layer shown in figure 2.3 also allows for the graphical development of models. Through the IDE, the Generic Graphical Advanced environment for DEVS modeling and simulation (GGAD) [8] allows the developer to use a graph-based representation to specify models hierarchy, interconnections and behaviors to automate model generation. At the execution engine level, various features have been implemented in order to improve the software including DEVSRT simulation algorithms, a Flattened Coordinator technique and a Time Interval function. The simulation algorithms allow correct handling of simultaneous events through the implementation of a messaging behavior for model interaction (See next section for details). The Flattened Coordinator technique 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.

E-CD++ has four main components [Figure 2.4]: the Main Runtime System, the Modeling Subsystem, the Runtime Subsystem and the Messaging Subsystem [4]. The Main Runtime System manages the overall aspects of the real-time execution and provides timing functions with microsecond precision. The Main Runtime System is the first object that is created in non-real-time context, and it launches the Runtime Subsystem [4] [8]. In general, the Main Runtime System first register Atomic component objects, then the Top coupled component ports that are connected to the external environment, reads in the external events (from an existing event-file) and builds an external event table. After that, the Main Runtime System reads in the model-file and builds the model hierarchy. Finally, it spawns the main real-time task in which the Root Coordinator (RC) is created to start the DEVSRT execution cycle.

The Runtime Subsystem consists of Simulators, Coordinators, and the Processor Admin. In E-CD++, simulators are run-time engines that correspond to atomic components, and they perform the main job of executing the internal transition and output function after receiving the proper messages. Coordinators are coupled models execution engines. The RC is a special Coordinator that manages the real-time event scheduling. It initializes the global Driver object which launches the real-time input driver tasks (which are associated with input ports of the Top coupled component in the DEVS model hierarchy) declared by the user.

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

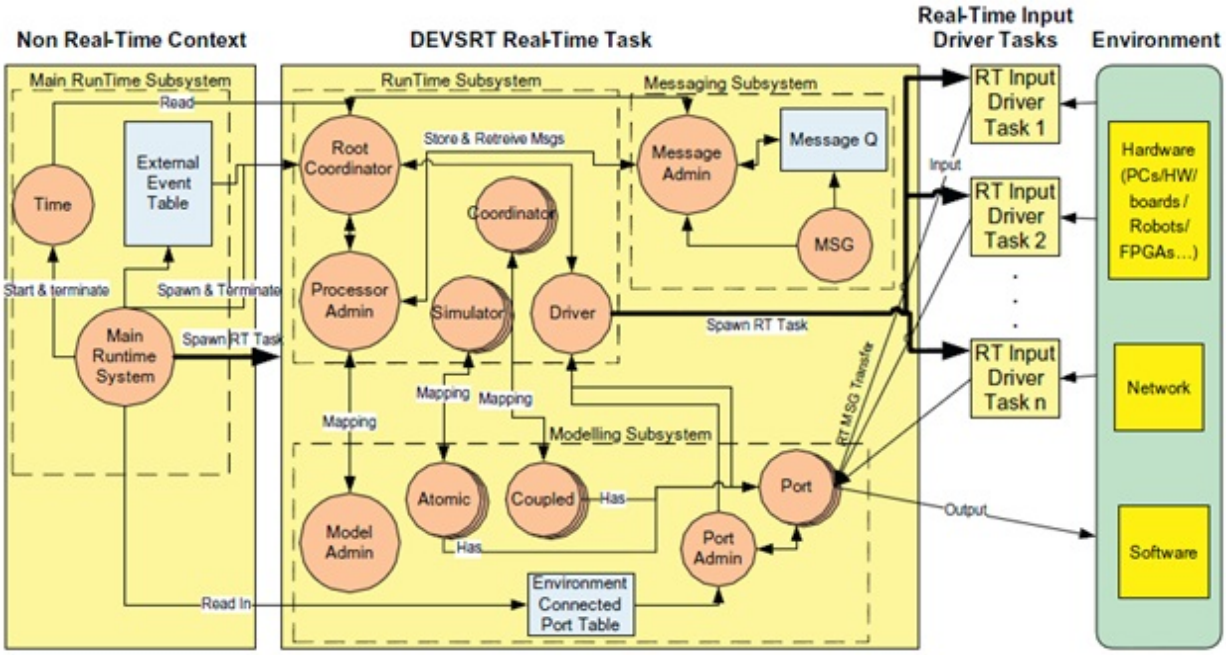

Figure 2.4: E-CD++ Software Components

behavior [8]. PDEVS models as well as their execution are explained in the following section.

2.3.2 PDEVS Model Definition

The original DEVS formalism was extended to resolve serialization constraints and allow simultaneous events, defining what is called Parallel DEVS (PDEVS) [98]. As with classic DEVS (Appendix A), systems are described using states that change upon the reception of an input event or the expiration of a delay. To tackle the complexity of the system, PDEVS decomposes the system into behavioral models called *atomic models* and structural models called *coupled models* similar to classic DEVS. Two types of events - internal and external events - influence the behavior of atomic models and cause state changes. When an external event occurs, an external transition function is executed and determines the new state of the model. In the

absence of external events, the model stays in a state s for a certain time *ta(s)*. When *ta(s)* expires, e.g. an internal event occurs, the model outputs a value and then changes to a new state given by an internal transition function. In classic DEVS, whenever two models are scheduled for state transitions at the same time, one of the models is chosen according to a *select* function provided in the coupled model specification. This function provides a tie-breaking mechanism by defining an ordering over all the components so that only one model is picked in the case of simultaneous events. Hence, collision behaviors cannot be represented properly and serialization is introduced in the execution of components. With PDEVS, atomic models provide an additional confluent function specifying what to do under such collisions, and it uses bags of events for receiving inputs and collecting outputs. Therefore, external and output functions handle bags of events allowing simultaneous processing of multiple events and eliminating the necessity for the select tie-breaking function used in classic DEVS.

A parallel atomic DEVS is specified as follows [98]:

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

Where X is the set of input events, S the set of sequential states, Y is the set of output events, δ_{ext} is the external transition function, δ_{int} is the internal transition function, δ_{con} is the confluent function, λ is the output function and ta is the time advance function. In this specification, the confluent transition function $s' = \delta_{con}$(s,e,x) computes the next state using the current state s, the elapsed time e and the input events x when the internal and external events occur simultaneously. The rest of the PDEVS specification follows the classic DEVS specification [7].

A parallel coupled DEVS is specified as follows [98]:

$$CM = \{X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}\}$$

Where X is the set of input events, Y is the set of output events, D is the set of the component names, M_i is the DEVS system of component name i ∈ D, I_i is the influences of i for each $i \in D$, and $Z_{i,j}$ defines the i-to-j output translation for each j in I_i. The whole CM specification follows the classic coupled DEVS specification except for the Select function.

The DEVSRT [4] formalism, used in E-CD++, is built on top of PDEVS and associates a deadline to each atomic model since meeting deadlines is crucial in real-time systems. A DEVSRT atomic model is formally defined as:

$$MRT = \langle X, S, Y, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta, d \rangle,$$

Where:

- X, S, Y, δ_{ext}, δ_{int}, δ_{con} and λ are the same as PDEVS.
- ta: S $\rightarrow$ $R^{+}_{0,\infty}$, time advance function which works with physical clock of the system
- d: S $\rightarrow$ $R^{+}_{0,\infty}$, is the relative deadline of each state for output production.

The coupled model definition is the same as PDEVS. Since DEVSRT is consistent with the PDEVS formalism, PDEVS models can be reused for RTES.

2.3.3 PDEVS Model Execution

Each DEVS specification can be executed by an abstract simulator that defines the execution semantics of the models. The PDEVS abstract simulation algorithms uses two kinds of components: Simulators (in charge of Atomic models), and Coordinators (in charge of Coupled models), using a one to one mapping between models and simulation components. Simulators are the engines that invoke the model transition functions (δ_{int}, δ_{ext}, δ_{con} , ta, λ); Figure 2.5 shows a coupled model example and its corresponding execution tree. Atomic models are associated with simulators (denoted by "S") while coupled models are linked to coordinators (indicated by "C"). For instance, S:B is the simulator calling the buffer (BUF) δ_{int} if an internal transition is needed. Coordinators on the other hand are in charge of event routing and hierarchical scheduling. C:B+P for instance will route events to S:B and S:P.

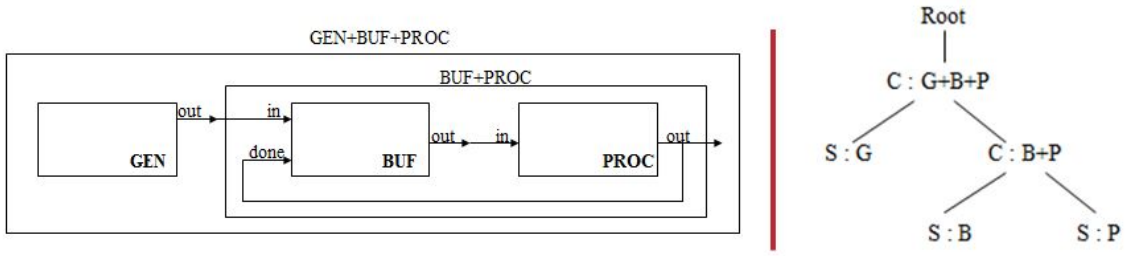

Figure 2.5: Models and Execution Engines

Simulation/Execution advances through message exchange. Five types of messages are used: three for synchronization (i.e. *,@,done) and two for content(q,y).

Synchronization messages and the actions triggered by each message are shown below:

- (@, t) : Output execution
- (*, t) : State transition

- (done , t): End of action

For content messages,

- (y , t) : Output event
- (q , t) : External event

Parents and children communicate via this message passing mechanism. The abstract simulator also provide a set of algorithms of the coordinators, and simulators that specify how each simulation engine reacts upon the reception of those messages. The simulators and coordinators implemented in E-CD++ communicate via this message passing mechanism, and follow the abstract algorithms explained in the next section.

2.4 From DEVS Models to Real-Time Execution

In this section, we will mainly outline the algorithms used to run the models and the real-time interface that allow the real-time models to interact with the surrounding environment.

Chow [98] defines the abstract simulator that defines the execution semantics of PDEVS models presented earlier. The simulation starts from a main loop which drives the whole simulation by repeatedly sending *(@,t)* and *(*,t)* to the topmost coordinator and waiting for a done message to advance the global simulation clock to tN.

Simulator Algorithms [98]

A *(*,t)* is used to synchronize three different transitions on the atomic model. *(@,t)* and *(done,t)* messages are used to invoke the output function before any transition

function as the output function depends on a state prior a transition at the same instance.

```
when a (@, t) message is received
        if t = tN then
                y := λ(s)
                send(y, t) to the parent coordinator
                send(done, t) to the parent coordinator
        end if
        else raise error
end when
```

Any content of an input message *(q,t)* is saved in a bag, and a done message sent to the parent coordinator.

```
when a (q, t) message is received
        lock the bag
        add event q to the bag
        unlock the bag
        send(done, t) to the parent coordinator
end when
```

The reception of a *(*,t)* messages indicates an internal or external event and will be processed according to the following (where tL is the last change time, t the current time, and tN the next scheduled event time):

```
when a (*, t) message is received
     case tL ≤ t < tN and bag is not empty
           e := t - tL
           s := δ_ext(s, e, bag)
           empty bag
           tL := t
           tN := tL + ta(s)
     end case
     case t = tN and bag is empty
           s := δ_int(s)
           tL := t
           tN := tL + ta(s)
     end case
     case t = tN and bag is not empty
           s := δ_con(s, e, bag)
           empty bag
           tL := t
           tN := tL + ta(s)
```

```
     end case
     case t > tN or t < tL
          raise error
     end case
     send (done, tN) to parent coordinator end when
end when
```

If the simulation time reaches tN (time of the next internal event), compute the output function and send the output events to the parent coordinator. If the bag is empty, only the internal transition takes place, otherwise, both internal and external functions take place at the same time leading to the confluent function to be executed. After this, the next tN is calculated and a done message sent to the parent coordinator.

Coordinator Algorithms [98]

The implementation of a coordinator is guided by the following algorithms:

```
when a (@, t) message is received from parent coordinator
     if t = tN then
          tL := t
          for all imminent child processors i with minimum tN
               send (@, t) to child i
               cache i in the synchronize set
          end for
          wait until (done, t)  s  are received from all imminent processors
          send (done, t) to the parent coordinator
     else raise an error
end when
```

Any output/input message is forwarded according to the coupling relations $Z_{i,j}$ to other simulators and coordinators. Note that the (y,t) messages will be processed within the wait statement when receiving a (@,t) message to guarantee that the outputs of any model are routed to their immediate influencees bags.

```
when a (y, t) message is received from child i
    for all influencees, j of child i
        q := z_{i,j}(y)
        send (q, t) to child j
```

```
        cache j in the synchronize set
    end for
    wait until all (done, t)  s  are received from  j  s
    if self is in I_i; (y is to be transmitted upward) then
        Y := y_{i,self}(y)
        send(y, t) to the parent coordinator
     end if
end when
```

Any content of an input message (q,t) is saved in a bag, and a done message sent to the parent coordinator.

```
when a (q , t) message is received from parent coordinator
    lock the bag
    add event q to the bag
    unlock the bag end when
end when
```

When an internal message (*,t) is received, (q,t) messages are sent to influences first and the bag is emptied . Then, (*,t) is forwarded to all components of the coupled model. This procedure [98] is shown here:

```
when a (*, t) message is received from parent coordinator
        if tL ≤ t ≤ tN then
                for all receivers, j in I_self and all q in bag
                        q := z_{self,j}(q)
                        send (q, t) to j
                        cache j in the synchronization set
                end for
                empty bag
                wait until all (done,t)'s are received
                for all i in the synchronize set
                        send (*, t) to i
                end for
                wait until all (done, t)'s are received
                tL := t
                tn := minimum of components' tN's
                clear the synchronize set
                send(done,t) to parent coordinator
        end if
        else raise an error end when
end when
```

In order to run models in a real-time context, the simulator time is tied to the underlying computing system. In a DEVS-based system, the simulation time advances

only when there is an event to be serviced; however with the real-time mode, the time-advance function is tied to the system clock. Therefore, the root coordinator only waits for the physical scheduled time of the next event to arrive and sends the appropriate simulation message.

In addition to using the physical time, DEVSRT utilizes model outputs to control hardware and provides a formal interface between the model and its environment. This is achieved by associating a driver object to the I/O ports of the top-most coupled model. The driver object is an abstract function that can be overwritten to adapt to different platforms and components.

To include the driver concept, the DEVSRT notation of the top coupled model in the model hierarchy is defined as follows:

$$TOPCM = \langle X, Y, OS, IS, DX, DY, D, \{M_d | d \in D\}, EIC, EOC, IC \rangle,$$

Where:

- X, Y, D, M_d, EIC, EOC and IC are the same as PDEVS .
- IS = is | is is the input signals from environment is the set of environment input signals.
- OS = os | os is the output signal to environment is the set of hardware output signals.
- DX: IS $\rightarrow$ Xv: converts external environment input signals to input port value (Xv).
- DY: Yv $\rightarrow$ OS: converts output port value to external environment output signals (Yv).

The RC algorithm in DEVSRT uses the above definition and is shown below. The root coordinator waits for signals from the environment or an internal timeout and then sends the appropriate message.

```
main():
forever for each DEVSRT model /* main loop */

wait for IS signals from environment or internal time out
if an external event then
        q = DX(IS)
        send (q, t) msg
        send (*, t) msg
else if an internal time out then
        send (@, t) msg
        send (*, t) msg
else if receive (y, t)
        OS = DY(y)
        send OS signal to the hardware
else if receive (done, t)
        tN = t
end if

end forever
```

Listing 2.1: E-CD++ root coordinator algorithm

Any communication between the environment and an atomic component goes through the top model. This allows hardware-in-the-loop and human-in-the-loop simulation by connecting DEVS models with hardware or humans.

With this approach, model continuity is integrated since the original models are finally deployed on the hardware where they act as controllers. The incremental hardware deployment made possible by this method offers a seamless integration mechanism, where provably correct software is embedded in the hardware, and formal interfaces for the communication between models and the environment. Moreover, the same models / source code can be reused for different target platforms with only the driver objects needing adaptation. The main goal is to directly deploy the developed control models as the final control software. While this technique supports model continuity, offers formal generic user implemented hardware interface, and includes

deadline specifications; there are limitations to overcome in order to be deployable onto a wider range of target platforms and promote fast prototyping.

Note: Some terms (e.g. scheduling, synchronization) used in this section, although similar to the operating system vocabulary, are to be understood in the DEVS execution context.

2.5 Hardware Execution Platform

In this section, we will review information related to the hardware target platforms on which models are executed. We will start by introducing the deployment platforms we chose, i.e. ARM microcontrollers.

2.5.1 ARM Microcontrollers

Advanced RISC Machines Ltd, ARM for short, designs processors and other components that are licensed to various silicon vendors. Today, the ARM architecture has become widely adopted, especially in 32-bit microcontrollers (MCU), in embedded industry. MCU manufacturers that have licenses ARM processor designs (IP licensing), adds to the ARM processor other design blocks like peripherals and memory since the CPU only takes a small part of the silicon area. Such microcontrollers built around ARM processors are often referred to as ARM-based microcontrollers or ARM microcontrollers.

Over the past few years, ARM has diversified its products and now offers multiple processor families. One of them is the Cortex family, divided in three profiles: A, R and M [99] [100]. First, the A profile is designed for high performance applications and used in devices such as iPhones that have to run high-end embedded OSes. Next,

the R profile is designed for the higher-end of the real-time market (e.g. automotive systems). Finally, the M profile is for smaller scale applications, fits criteria such as low cost, low power, low interrupt latency and energy efficiency, and provides deterministic behavior required in many real-time controller systems. It is ideally suited for microcontrollers, used in consumer products (e.g. toys, electrical appliances) and even industrial and medical systems.

The devices that we used (shown in Appendix B) are built around Cortex-M3 [101] and Cortex-M4 [102] processors, two products from the ARM Cortex-M family. Both the Cortex-M3 and Cortex-M4 are broadly used namely in microcontroller products, System on Chips (SoC) and Application Specific Standard Products (ASSP) [99]. Cortex-M3 and M4 also have the great advantage of being compatible with a wide family of other ARM devices. This allows easy migration between ARM devices.

In terms of general technical specifications, all the ARM Cortex-M processors are 32-bit RISC (Reduced Instruction Set Computing) processors with 32-bit registers, 32-bit internal data path and 32-bit bus interface. They also support 8 and 16-bit data. Cortex-M3 and M4 particularly support some operations (e.g. accumulate, multiply) that involve 64-bit data. The two processors also have a three-stage pipeline design (instruction fetch, decode, and execution), and both have a Harvard bus architecture, which allows simultaneous instruction fetches and data accesses. Many similarities particularly exist between Cortex-M3 and M4. They both contain a core processor, a Nested Vector Interrupt Controller (NVIC), a SysTick timer, internal bus systems, a Memory Protection Unit (MPU), and components to support software debug operations. Only the Cortex-M4, however, is equipped with a floating-point unit. The Cortex-M4 is able to deliver higher performance in DSP applications, supports floating-point operations, and executes some instructions in fewer clock cycles.

The processors do not include memory (i.e. program memory, SRAM or cache)

per se but include a generic on-chip bus interface to allow microcontroller vendors to add their own memory system and components. Apart from the processor, the rest of the silicon area of a MCU is usually occupied by program memory (e.g. flash memory), data memory (typically SRAM), peripherals, internal bus infrastructure, a clock generator, I/O pads, analog components such as ADC and DAC, and support circuits for manufacturing tests. Although these components may greatly vary depending on the vendors, the main concepts to know, in order to use a Cortex-M based MCU, are how the processor works and what peripherals are available on the MCU. Indeed, in most cases, the processor controls the peripherals and handles the system management both accessible from the memory map. Besides, MCU vendors usually provide C header files and driver libraries to ease the software development task.

2.5.2 Hardware Peripheral Libraries

As previously mentioned, microcontroller vendors provide header files and C code that include the definitions of peripheral registers, and functions for peripherals configuration and access. Because creation of software is a major cost factor in the embedded industry, ARM has released CMSIS [103], the Cortex Microcontroller Standard Software Interface, to provide a common interface for the Cortex M processor series and allow microcontroller vendors to have a consistent software infrastructure. Consequently, most Cortex-M MCU vendors provide device libraries based on CMSIS which provides a small abstraction layer between the microcontroller and the rest of the code. STMicroelectronics — the vendor of the Disco and Nucleo boards (Appendix B) and manufacturer of the eval board MCU — offers multiple device libraries solutions with different abstraction and portability levels as illustrated in Figure 2.6.

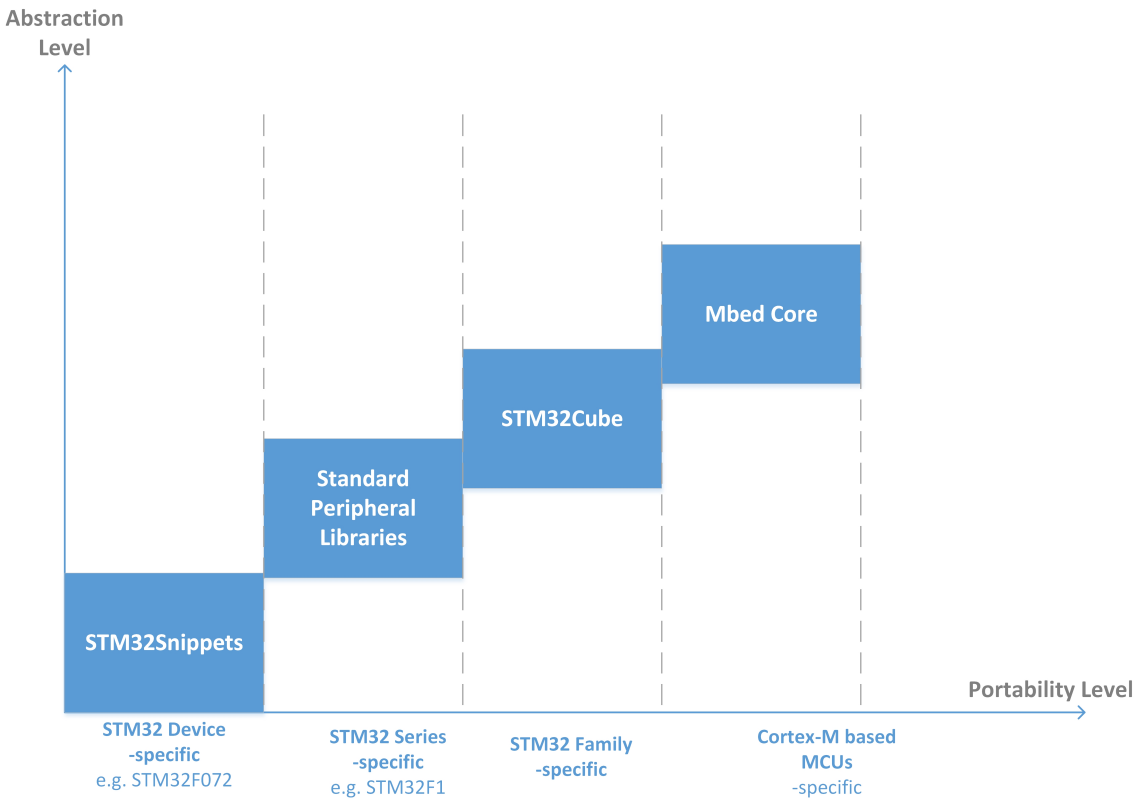

Figure 2.6: STMicroelectronics Device Libraries - Abstraction and Portability

Available solutions include:

- STM32 snippets [104]: These are collections of highly optimized device-specific code examples based on STM32 peripheral registers. They are suitable for low level embedded system developers, used to assembly or C with little abstraction. Snippets are however not portable directly between series, and only limited to L0 and F0 devices.

- Standard Peripheral Libraries (SPL) [105]: These are C libraries covering STM32 peripherals and reusable within the same series (F2 for example). They offer a higher level of abstraction compare to snippets and are ideal for embedded systems developers with procedural C background. We used the SPL to run E-CD++ on the Eval board.

- STM32Cube [106]: STM32Cube enables quick and fast development with STM32 devices, and provides high portability inside the entire STM32 family. STM32Cube particularly features a graphical software configuration tool, the STM32CubeMX, that allows the user to graphically access and configure MCU peripherals through graphical wizards. Corresponding code is then generated and can be included in the embedded project. It allows the developers to save time usually dedicated to MCU configuration and makes the development task easier.

The generated code relies on HAL libraries (STM32Cube HAL) that were designed to provide a generic interface that spans the entire STM32 family and standardizes peripheral access. STM32Cube also include a set of middleware components for applications based on USB, TCP/IP and graphics.

2.5.3 MBED, Rapid Prototyping and IoT Platform

In the previous section, we have seen how STMicroelectronics progressively introduced device libraries that span the entire STM32 family, i.e. their 32-bit MCU. Here we will present an initiative — MBED — aimed at providing a common solution to ARM Cortex-M devices. ARM and its technology partners develop the MBED project collaboratively and target the microcontroller applications, Internet of Things and Wearables markets.

MBED offers an online code editor and compiler, a Software Development Kit (SDK) and a Hardware Development Kit (HDK). The SDK provides the MBED C/C++ software platform and tools for creating microcontroller firmware that runs on smart devices. It mainly includes core libraries for the MCU peripheral drivers, networking, RTOS and runtime environment. In order to support a larger range of

devices, and not be limited to STM32 MCUs, we have included some of the core MBED libraries to our bare-metal framework development. We will therefore take a closer look at these internal libraries that provide a vendor independent API.

Figure 2.7 [107] shows the design overview. The first three layers (*mbed API*, *mbed common*, and *mbed HAL API*) are microcontroller independent, meaning that they provide function definitions that span multiple products.

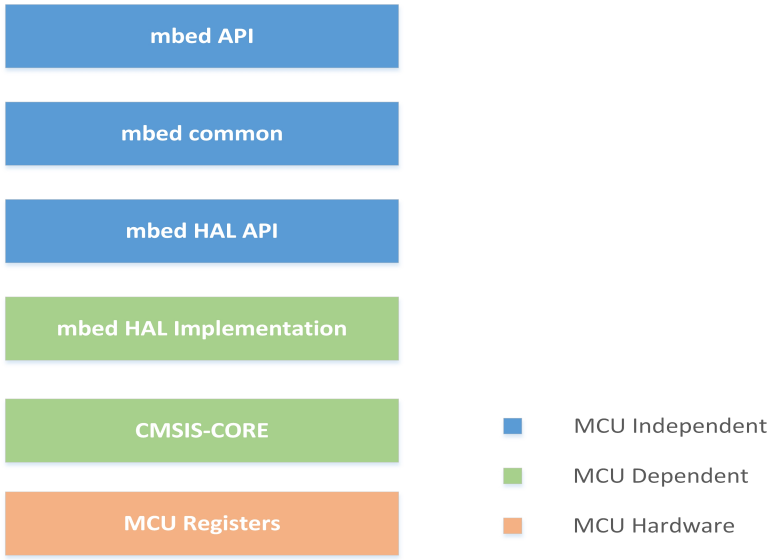

Figure 2.7: MBED Design Overview

The MBED HAL implementation and CMSIS-Core layers are MCU dependent. The first one is usually from a specific vendor and can be the STM32 HAL libraries for STM32 devices for instance. One of the main benefits of MBED is that the code can be exported from devices to devices without any change in the code. Only the target field needs to be updated so that the appropriate MCU dependent layers are included; the rest remain the same. In addition, the MBED API provides a friendly object oriented interface for fast prototyping.

Chapter 3

Problem Statement

The problem tackled in this thesis is standalone DEVS-based firmware development. The objective is to be able to execute models directly on the target system hardware without the need of an operating system. Three resolutions were made at the beginning of the project that narrowed its scope. The first one was to leverage the existing functionalities of E-CD++ in order to run it directly on the target platform without the need of an RTOS kernel like Xenomai. No new schedulability algorithms were included. Rather, we chose to focus on an effective message passing mechanism between model processors by introducing a new real-time executive based on a sequential PDEVS architecture. Finally, to manage the hardware resources properly while allowing rapid prototyping, we decided to introduce a hardware abstract layer that allows users to develop easily drivers for their applications. In addition, we developed a set of applications that run on top of the developed bare-metal kernels. A more accurate description of the problem would be the development of two bare-metal DEVS-based kernels and a set of application that runs on top of the newly designed kernels.

Prior to this thesis, it is improbable that DEVS real-time executives were run on bare-metal. Indeed, DEMES portrays the real-time executive as running on top

of an RTOS (Figure 1); and as reviewed in chapter 2, existing DEVS-based tools required some real-time operating system, kernel or virtual machine to execute models on hardware. As outlined in section 2.2, real-time extensions for the Linux kernel were namely used in PowerDEVS and E-CD++. These approaches pose problems since high performance microcontrollers are needed (including powerful processors, memory, and in many cases, secondary memory in order to allow the software stack to be executed without issues). Hence, although the DEMES approach offers multiple benefits, tools have to be improved to overcome limitations and support different hardware.

Our work extends the applicability of M&S driven development by providing OS independent DEVS real-time executives. As a result, target devices such as low power/memory microcontrollers where an operating system would require excessive resources, are now covered. In addition, certain systems and programs, given the trade-off between OS services and performance, err on the side of performance, as they need to run as fast as the technology allows. In the latter case, not all OS services are needed, and those that are needed can be included for the particular application. With our approach, we implement only the services, needed to run DEVS model; thus, eliminating extra unused OS features usually added by the use of a third-party RTOS - that could be costly in resources and/or introduce overhead. Consequently, this project further narrows the gap between the simulation and implementation phases by enabling the utilization the same models for both simulation and execution in limited resource and high-performance requirement environments. It also results in a decreased kernel footprint, increased efficiency and enhanced portability.

Since RTES are pervasive and varied, we believe that solutions addressing current development shortcomings should be applicable to a wide range of devices in order to properly replace traditional techniques and be appealing to industry. We

aim at providing a modular, component-based approach that addresses heterogeneity and complexity at higher levels, allows hardware software co-design, permits formal verification, and offers a unified and consistent development environment. Fostering fast prototyping is also an objective given the increasing number of new market applications.

In the following chapters, we will present kernels based on a formal M&S methodology that enables the user to run models directly on bare-metal. The new real-time executives presented here provide functionalities similar to those of a real-time kernel, with formal models operating as system processes. Based on the PDEVS abstract algorithms seen in section 2.4, scheduling is done by coordinators while simulators dictate the transition functions to be executed in order to produce the user specified behavior in the original models. By tying the clock to a physical time, real-time functionalities can be provided, and real-time operating system services covered. We add input/output and hardware resources management to the DEVS core in order to run n bare-metal. For the applications built in this project, we use ARM-based microcontrollers to test the new kernels. The ARM architecture is the most pervasive 32-bit architecture and is found in all types of computing devices from real-time safety systems (automotive braking systems) to smartphones [108].

Chapter 4

Bare-Metal Real-Time Executives

One of the key steps of the DEMES approach is the transition from the simulation platform to the target hardware. To enable this passage, a DEVS real-time executive runs DEVS models on the execution platform and allows them to ultimately control the real system. In this chapter, we will present two bare-metal real-time executives designed to address the problem stated in chapter 3. The first executive, referred to as the "legacy compatible version", preserves the same interface as CD++ and the previous version of E-CD++ (which is based on Xenomai). The second executive is based on sequential PDEVS algorithms and uses a different message passing mechanism. Before providing the details of each real-time executive, we will briefly review the main limitations of the previous DEVS real-time executive, the proposed approach used to overcome them as well as the implications of going bare-metal.

In the previous E-CD++ version (Figure 2.3), the runtime objects were running on top of the Xenomai real-time kernel as real-time tasks. Although a RTOS like Xenomai provides convenient services, it imposes restriction on memory capacity, processing, and portability as the target platform must include the memory and processing power necessary for the OS. Plus, the OS kernel must be compiled for the target platform and should interface with the available hardware devices.

Porting a Xenomai-dependent E-CD++ to embedded platforms with small memory capacities becomes therefore complicated by the application size as well as the Xenomai kernel it runs on. Solutions used to circumvent this shortcoming include the addition of external memory or the use of a network connection to interface with the target platform from a simulation platform. Indeed, E-CD++ includes a Telnet2Target feature [90] that may be used to send commands over a network interface for restricted memory devices. In this case, the simulation is run from a host computer; only a small driver program is embedded on the target and commands may be sent through Bluetooth or USB. Still, these solutions introduce several latencies in accessing memory and network communication delays, and require the main program to be run from a more powerful computer.

Besides, the dependency on the Linux kernel and the use of network drivers also reduce the portability of E-CD++ between different execution platforms. In fact, the target platform has to be chosen based on the availability of a kernel variant for this particular hardware, or the presence of suitable network interface to allow communication with E-CD++ running on a host computer.

As a solution to these limitations, we propose the new architecture shown in figure 4.1. With this approach, the development environment is used to define models, run tests, debug hardware and software, and deploy the resulting software onto the target platform. The final models run on the target platform without the need of a host computer, or real-time operating system middleware.

First, the modeler defines models using the DEVS formalism and C++ code; an Eclipse IDE is used in order to make the development task easier. These models are then interpreted and executed by a real-time executive that directly rests on the target platform. The driver interface layer is used to provide a formal interface for

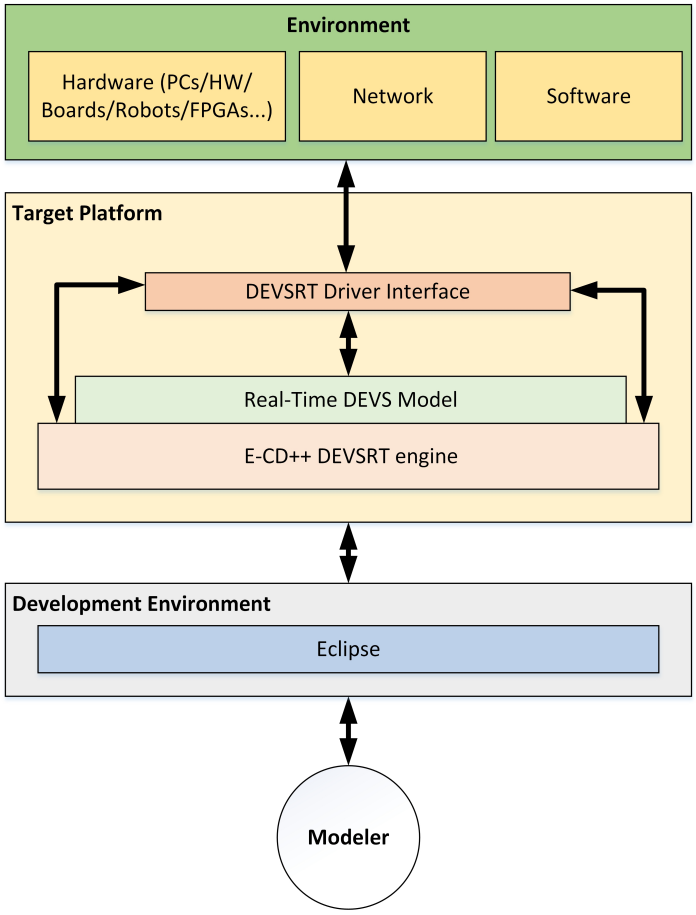

Figure 4.1: Bare-Metal E-CD++ Layers

I/O ports. However, this layer has been modified to communicate directly with the underlying hardware without the need of a real-time kernel middleware.

The bare-metal real-time executives we defined follow the concepts illustrated in figure 4.1 but differ in the model definition format and the model execution engines. These will be presented in the sections 4.2 and 4.3 .

4.1 Execution on Bare-Metal

In a broader view, porting a RTOS application to bare-metal has several implications as illustrated in Figure 4.2. The most left side shows an example of an application

running on top of a Linux real time kernel: The application uses the GNU Library - which provides the system call interface that connects to the kernel and the transition mechanism between the user and kernel spaces - and makes system calls to request different OS services. These OS services are provided by the Linux RT kernel, which serves as a middleware between the application and the hardware, and is in charge of managing and communicating with hardware.

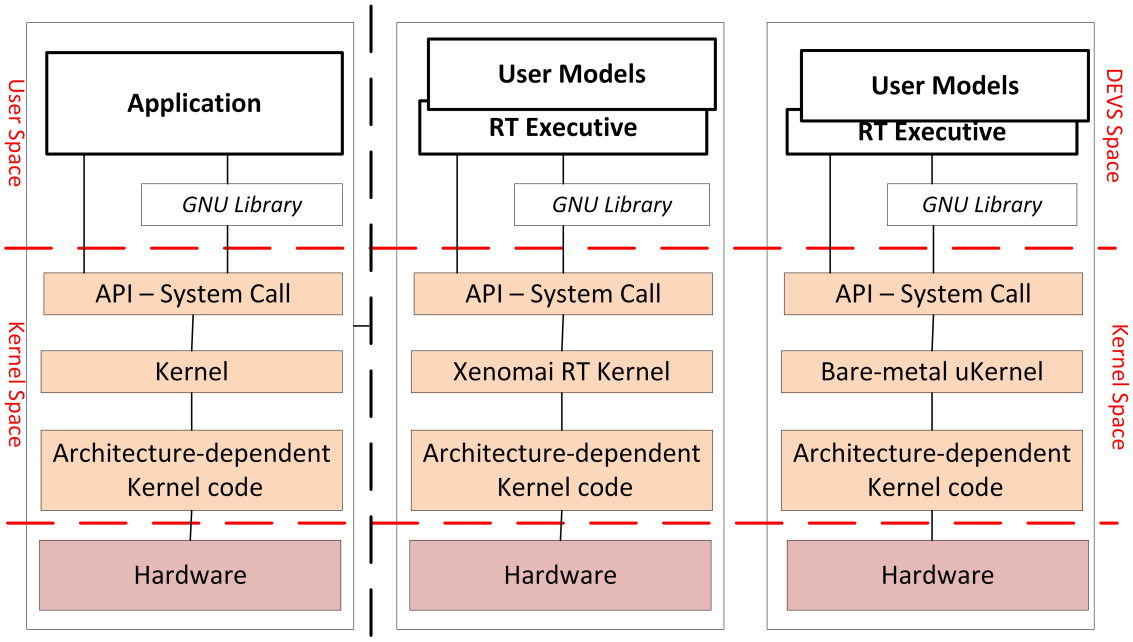

Figure 4.2: RTOS vs Bare-Metal - An Overview

The middle part of Figure 4.2 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 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. The architecture dependent kernel and the hardware layers will be presented in chapter 5; the new real-time executives will be described in section 4.2 and 4.3. We will first present the microkernel that services both real-time executives.

4.1.1 Removing OS Dependencies: From Xenomai to Bare-Metal

To successfully run on bare-metal, all operating system dependencies needed to be removed and a microkernel developed in order provide the essential services previously offered by the Xenomai real-time kernel. The microkernel should mainly provide basic file, memory, and hardware resource management.

An operating system usually manages hardware and software resources and sits between the application and the hardware. An application commonly request OS services by system calls. For the bare-metal real-time executive, we identified the requested services through system calls tracking and created functions with the same signature but with a re-designed implementation that takes into accounts the limitations and environment of the target platform. These functions provide file and input/output management and reply to system calls such as open, read and write. These are necessary since E-CD++ needs to read the user-model file that specified the coupled model.

System functions related to memory management (e.g. sbrk) are needed for allocation/deallocation of memory. This required the run-time modification of pointers to heap memory allocation. Indeed, E-CD++ is developed in C++, an object-oriented language, and dynamic memory allocation is required in order to allow for the instantiation of new objects.

Other system services associated with inter-process communications within a multi-processing system are unnecessary. For instance, the *getpid* function to return the process ID of the currently running process. As there is only a single process running, the value returned by this function can be set to an arbitrary integer that meets the constraints of what would be expected from an application launched from an OS. Multi-processing and multi-programming can be implemented directly at the model level and natively handled by the execution module based on PDEVS algorithms. In this context, models act as processes, and the RC as the scheduler. Periodic and aperiodic actions can be managed with timers and interrupts.

To offer other useful services generally provided by the OS, we need to provide similar functions to replicate them. This was achieved by using hardware components such as the real-time clock, on-board memory and low power modes. Startup code, low-level initialization, linker script and interrupt handling mechanisms were also developed to allow the bare-metal execution. With all the above changes, the replication of key OS functionalities and complete removal of the OS becomes possible. The new functions were developed to provide essential functionalities requested through system calls without the overhead of a full OS kernel. The following sections will present the real-time executives that run on top of this newly developed microkernel.

4.2 Embedded CD++, the Bare-Metal Version

The bare-metal version of E-CD++ was designed to be compatible with CD++, and allow previous models to be executed on bare-metal and run on new target platforms. This version preserves the existing CD++ interfaces in terms of model definition and driver specification.

Modifications were made to the real-time executive in order to provide stand-alone operation. To achieve this, multiple functionalities were leveraged and new ones added. We will start by presenting the new software architecture, explain the functionalities adaptations, then present the alterations made to the existing subsystems, and finally present the hardware interface.

4.2.1 Software Architecture

Similar to the Xenomai version, the core components of E-CD++ include the main runtime subsystem, runtime subsystem, modeling subsystem and the messaging subsystem. Figure 4.3 shows these subsystems and their interaction in the bare-metal version. There are three main differences compared to Figure 2.4 (Xenomai-based E-CD++):

- All the components rest directly on the target platform
- A flattened coordinator is used instead of multiple coordinators
- No Xenomai real-time tasks are present for the main DESVRT task and input driver tasks.

The main runtime subsystem manages the global aspect of the real-time execution and provides timing execution with a precision of one microsecond. In the previous version, this was done by incorporating Xenomai native skin clock functions in the E-CD++ Time class; for the bare-metal version we use a 32-bit hardware timer and the onboard microcontroller clock to obtain the same precision. The main runtime subsystem is also in charge of loading models and ports, constructing the models and processors hierarchy, loading external events and initializing the root coordinator.

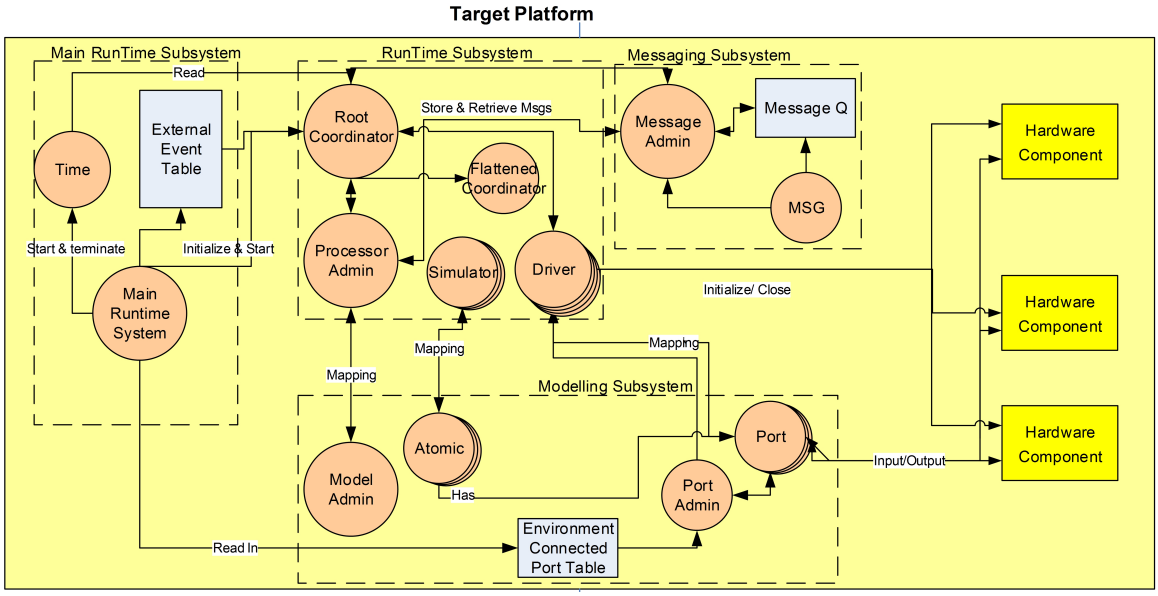

Figure 4.3: Software Components

Contrarily to the previous version, the RC is not spawned as a real-time task, since control is then passed to the root coordinator. Once initialized, the root coordinator manages the rest of the model execution by sending the first message that triggers other processors to send/receive messages, and handling real-time event scheduling. It also manages the global Driver object that handles the user defined input/output port drivers. These latter are associated to the hardware components (e.g. sensors, actuators, serial peripheral interface...). The global driver object calls the defined port driver functions to get or set the port value and manages related input/output DEVS messages, therefore serving as a bridge between ports and the RC. The Xenomai real-time task communication that was used between input driver tasks and the RC has been replaced by a combined polling interrupt driven mechanism. The RC periodically get inputs from hardware components by invoking the global driver while waiting for the next internal transition to occur, sends appropriate messages in the occurrence of an external event or internal timeout, and also send outputs to the output hardware components. The RC, along with the flattened coordinator,

simulators, drivers, and the processor admin are part of the runtime subsystem. The Flattened Coordinator, in particular, was used to minimize the number of message passed between models, since embedded devices generally lack memory and processing power. This is achieved during model loading by redirecting and removing the links between coupled models and establishing direct links from the top models to the atomic models - the flattened coordinator is therefore directly linked to simulators in the processor hierarchy. In order to be executed, the defined models are associated to a processor. Traditionally, each atomic model is associated with a simulator, while coupled models are linked to coordinators. The bare-metal version uses a flattened coordinator that directly manages simulators without the need of extra coordinators (illustrated in figure 4.4). The flattened coordinator reduces the number of messages passed between models. This is accomplished through the removal of coordinators for coupled models, which are replaced with a single flattened coordinator that manages the message passing between atomic models enabling direct communication and reducing exchanged messages.

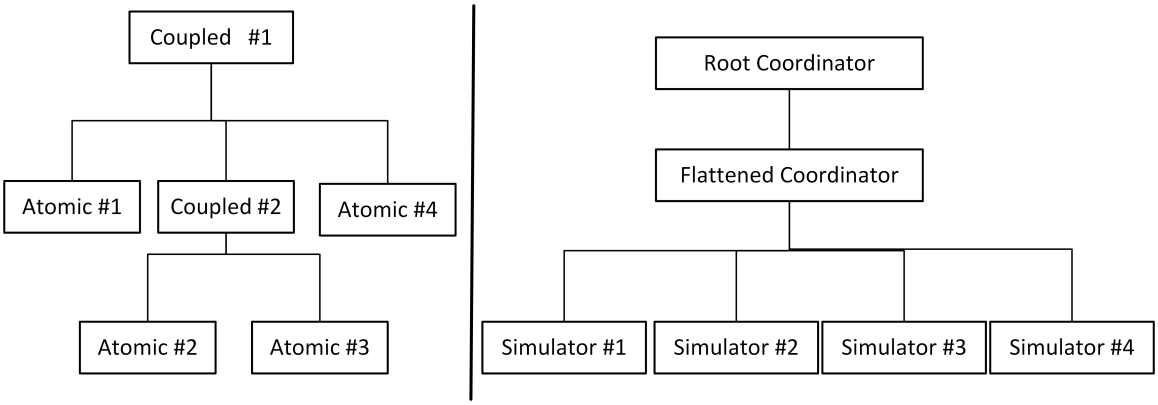

Figure 4.4: Model and Processor Hierarchy with the Flattened Coordinator

The flattened coordinator technique improves processing power and speed, which is a limitation to execution on embedded platforms. In order to increase efficiency, the

Flattened Coordinator analyzes the links between models on initialization and generates an influence list, establishing the relationships between models. The flattened coordinator is able to identify the recipient of a message and passes the message directly to the atomic model. In a simple system containing only a few coupled models, this will not have a very large effect on the overall efficiency of the system; however, as the system complexity increases, the increase in performance that is achieved through the implementation of the Flattened Coordinator technique can be seen to improve [109].

The messaging subsystem includes the message admin and various message classes corresponding to the ones defined in the PDEVS abstract algorithm. Messages are sent to processors (simulators and coordinators) through the message admin that communicates with the processor admin. Incoming messages are first stored in the message queue and then processed by the message admin.

Finally, the modeling subsystem holds the model hierarchy defined by the user through the specification of atomic and coupled models. Note that coupled models do not appear in the figure 4.3 since they are eliminated as the flattened coordinator is systematically used for the bare-metal version. Apart from model classes, the port admin contains a list of top ports or ports connected to hardware components. These are later used by the global Driver object to communicate with hardware components.

In the following section, the above main changes will be detailed and their implementation further explained. We will start the modeling and runtime subsystems, the main runtime subsystem and finally the hardware components interface.

4.2.2 Modeling and Runtime Subsystems

The modeling and runtime subsystem are the main components that implement the PDEVS formalism. The modeling subsystem contains elements necessary for model

construction while the runtime subsystem holds the model execution mechanism. These subsystems are respectively built around two main abstract classes: *Model* and *Processor*. *Model*, on one hand, is used to define the behavior of atomic models and the structure of coupled models. *Processor*, on the other hand, is used to execute atomic and coupled models through simulators and coordinators. The root coordinator is a special coordinator associated with the top model and oversees the simulation. The previous entities are shown in figure 4.5.

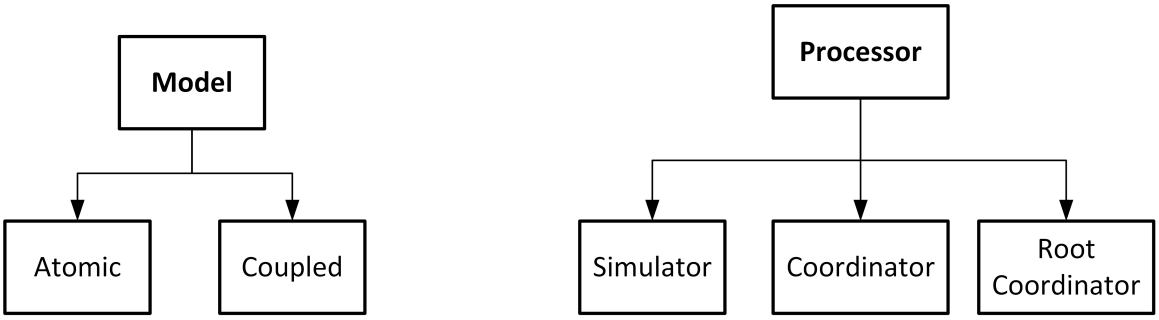

Figure 4.5: Models and Processors

The class hierarchy shown in figure 4.5 is implemented in CD++ and in the previous E-CD++ with an optional flattened coordinator. Note that both models and hierarchy trees are constructed by the main runtime subsystem before invoking the root coordinator. We will first talk about the implementation of models and then in the context of the bare-metal version.

The users define the structure of the models in a model-file using a special format defined in [95]. The model-file contains the components (Atomic and Coupled) in hierarchical order, in which the top-most coupled component is declared first. For each coupled component, its internal components (atomic and/or coupled models), input/output ports, and links i.e. External Input Couplings (EIC), External Output Couplings (EOC), and Internal Couplings (IC) are declared. A new internal program converts this model file into hexadecimal in order to be embeddable on the target

platform and part of the final executable. For the behavioral part, each atomic model is derived from a basic atomic class, and provides state transition functions (i.e. internal, external) and output function implementation.

As for the processors algorithms, simulators and flattened coordinator implement the PDEVS abstract algorithms presented in section 2.4. The root coordinator implements the same DEVSRT algorithm, except that obtaining signals from the environment is achieved through a different strategy. Indeed, in the Xenomai version, a mailbox is used for the communication between input driver tasks and the RC task. When waiting for an internal event timeout, the RC would wait to receive a message from a Xenomai real-time task indicating hardware input. Since Xenomai real-time tasks are no longer available, the reception of the done message will cause the RC to sleep until the next internal transition is scheduled, periodically verifying that an external event has occurred when in interrupt mode. If an external event occurs, the event will be processed prior to the internal transition and the cycle will be repeated. In the case where there are no more internal transitions scheduled, the Root Coordinator will place the microcontroller into a low power mode and await an external event. Note that there is also a polling mode option that periodically calls input port driver functions to verify if events have occurred. External events are gathered through ports and drivers and will be explained in the next section.

Hardware device connections and interfaces are managed through the use of two classes: *Port* and *Driver*. The Port class belongs to the modeling subsystem and enables the user to specify ports connected to hardware components. They also provide functions to convert external environment signals to input port values or output port values to external environment output signals. The driver class is the corresponding execution mechanism and belongs to the runtime subsystem. It is invoked by the RC to get signals from the environment or send output values to

hardware components. Together, *Port* and *Driver* provide a link between the model implementation, and the hardware target platform.

The Port class represents the logical connection between models and hardware devices. Instead of passing established API commands over a network interface (as with the libplayerc) through the port class as in the Xenomai version, the bare-metal implementation allows input and low-level functions to directly access different hardware components. A hardware API was also included to allow rapid prototyping for ARM microcontrollers and will be presented in chapter 5. When a signal is detected on an input port, a corresponding PDEVS message (a *(q,t)* message to be specific) is generated and added to the message queue. When a port is configured as an output, the port receives data from the associated driver in charge of converting the received PDEVS message (an output message - *(y,t)*) from the RC into suitable port data. One of the advantages of the bare-metal approach, especially for experienced embedded systems developer, is the option to use of hardware or software interrupts to directly detect changes on the hardware components and generate corresponding PDEVS input messages. Indeed, specific hardware interrupts associated with each hardware device can be used to signal an input event while software interrupts can be programmed based on a division of the base clock to provide periodic polling. Interrupt service routines are then set to post a port value that is then used by the port driver to generate a PDEVS message.

In the Xenomai version, Driver objects are associated to periodic real-time tasks that alternately post hardware event to the RC task mailbox. The RC then pends on the mailbox while waiting for an internal timeout and processes the content of the mailbox upon reception of a message. This technique can cause overheads due to task context switching especially when driver tasks have a short period. With the bare-metal implementation, drivers are by default associated with timers similar to

real-time task periods and checked for events by the root coordinator when pending for an internal timeout, or interrupts for embedded developers accustomed to this concept. Alternatively, the role of the Driver class is to convert PDEVS output messages, initialize and close hardware devices at the beginning and end of execution. As mentioned, when an output message is received by the RC, the Driver reads the value from that message and passes it on to its associated Port for interpretation and communication to the device. In the case of initialization or termination, the Driver class includes functions that interface with hardware devices in order to prepare them for operation, or for the end of simulation as required.

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

4.2.3 New Main Runtime Subsystem

Since the main runtime subsystem is the first object to be created, it is in charge of initializing the system timing. This is achieved with a 32-bit timer set to trigger at 1MHz therefore providing microsecond precision as in the Xenomai-based version. This technique however imposes that the microcontroller's clock frequency to be at least 1MHz, which is common nowadays.

Apart from the clock initialization, the main runtime system also needs to register the model file on the bare-metal versions. Indeed, the only file referenced by E-CD++ during execution (and thus needed on the target platform), is the model file. Models are loaded into E-CD++ at run-time through the reading and interpretation of a model file. This used to be done by providing E-CD++ with the name and location of the model file from the command line. Since we do not have a directory structure for OS file I/O support, it was necessary to develop a pseudo file system in order to maintain continuity between desktop simulation and target simulation. In order to mimic this behaviour, the model files are loaded directly into memory and the file names are used to populate a file register. The file register then determines the memory address of the text file using a file table 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 two tree-like structures: the model hierarchy tree and the 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. Because the 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. This results in the hierarchy shown on the right side of figure 4.4. 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 as per the DEVSRT RC algorithm (listing 2.1).

4.2.4 Hardware Components Interface

In the older version of E-CD++, one of the strategy that was used to interface with hardware with constrained memory (e.g. Lego robots, ePuck) was the use of the "libplayerc" library, based on Player - a single device server that runs on the robot providing control over the sensors and actuators. This enabled the main program to run from a host computer and send commands or read data through the network interface provided by the player library. With the bare-metal, the footprint was reduced enough to hold onto some of those devices; 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. There is no networking

capabilities restrictions anymore.

Overall, a high level of portability and model continuity can be achieved, as the DEVS model is not changed throughout development. This design is also portable as 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.

4.3 Embedded CDBoost, a Sequential PDEVS-based Real-Time Executive

We will now present the second bare-metal real-time executive, Embedded CDBoost (E-CDBoost for short). This later is derived from on a new DEVS simulation software called "CDBoost" [9]. We will present common core components to CDBoost and Embedded CDBoost as well as the additions implemented to allow real-time execution on bare-metal. One of the major differences between CDBoost and the original CD++ is the simulation mechanism used to implement the processors. Although both arise from the PDEVS abstract algorithms presented in section 2.4, CDBoost uses a function call & return technique instead of the default message passing mechanism in order to achieve high performance. It implements new sequential algorithms to run PDEVS models and offers a modular and flexible architecture. In terms of implementation, CDBoost relies on the popular Boost C++ library [110] and follows the C++11 standard.

4.3.1 Software Architecture Overview

Figure 4.6 shows the architecture overview of CDBoost. Similar to CD++, it separates the model construction logic from the simulation mechanism. Model classes provide the former while execution classes implement the latter. Utility classes provide useful functionalities to the simulation such as time classes, message classes, input stream for external events and a future event list (or FEL) comparable to a scheduling structure.

Model classes contain three main classes: *Model* that offers a common interface to atomic and coupled models, *PDEVSAtomic* that can be extended to implement user defined atomic models, and *PDEVSCoupled* that provides an interface to specify the structure of a model.

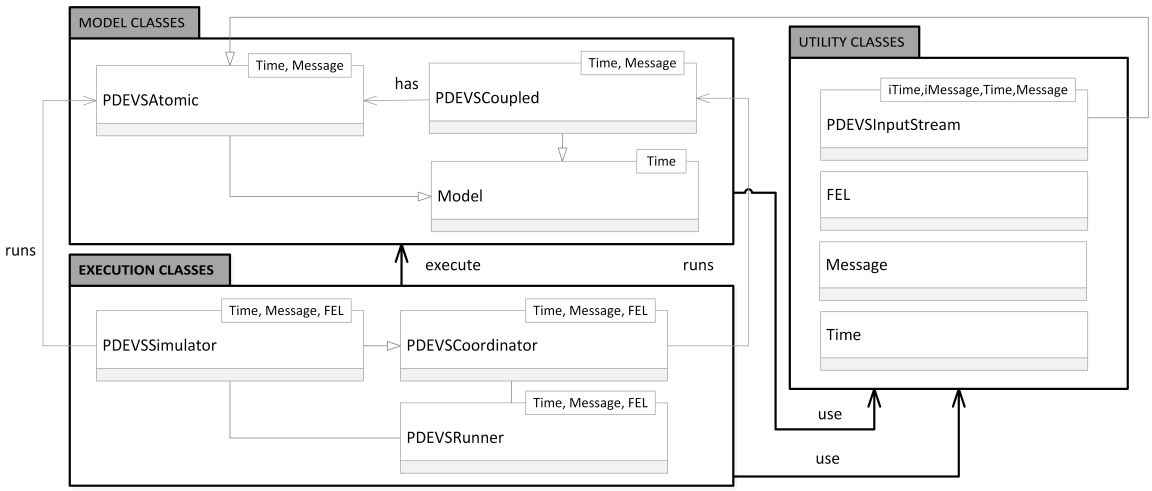

Figure 4.6: CDBoost, Software Components Overview

Execution classes, on the other hand, group *PDEVSSimulator* - to render atomic models behavior, *PDEVSCoordinator* - to execute coupled models, and *PDEVSRunner* similar to the root coordinator.

CDBoost was designed to run general simulations on a workstation, generating only simulated results. Embedded CDBoost (E-CDBoost), instead, is designed to

execute models on embedded hardware. This requires real-time extension and interaction with the environment. With E-CDBoost, inputs can be retrieved from hardware components such as sensors, timers, or data collected from human interaction. E-CDBoost outputs can actuate motors, valves, gears and other components. E-CDBoost also supports the integration of both simulated and real components for a facilitated hardware-software co-design. Since E-CDBoost is designed to execute in real-time (contrary to CDBoost that only allows as fast as possible simulation and advances time directly to the next event) and rest on embedded hardware, it includes a wall-clock time (with a microsecond precision) interfaced with an onboard hardware clock. Actions therefore happen in real time.

To the architecture shown in figure 4.6, E-CDBoost adds a *Port* component to the modeling subsystem, a *Driver* and a new *Runner* (ERunner) to the execution subsystem to interface model implementation and hardware platform. It also adds real-time functionality by replacing the virtual-time used in CDBoost, with physical time based on a hardware clock. This is achieved by defining a special Time class. The message structure is also adapted to the real-time environment and carries port and value information.

Before detailing each subsystem, we will first discuss the communication mechanism used between model and execution classes and the new sequential algorithms.

4.3.2 A New Execution Mechanism

CDBoost replaces top-down messages by function calls and bottom-up messages with return/replies. Figure 4.7 further illustrates the distinction between CD++ execution mechanism and CDBoost is illustrated. In the first case, nodes in the processor hierarchy tree communicates by sending different messages and executing actions locally while functions (*advance_simulation*() and *collect_outputs()*) of lower nodes in

the hierarchy are called and values returned in the second case.

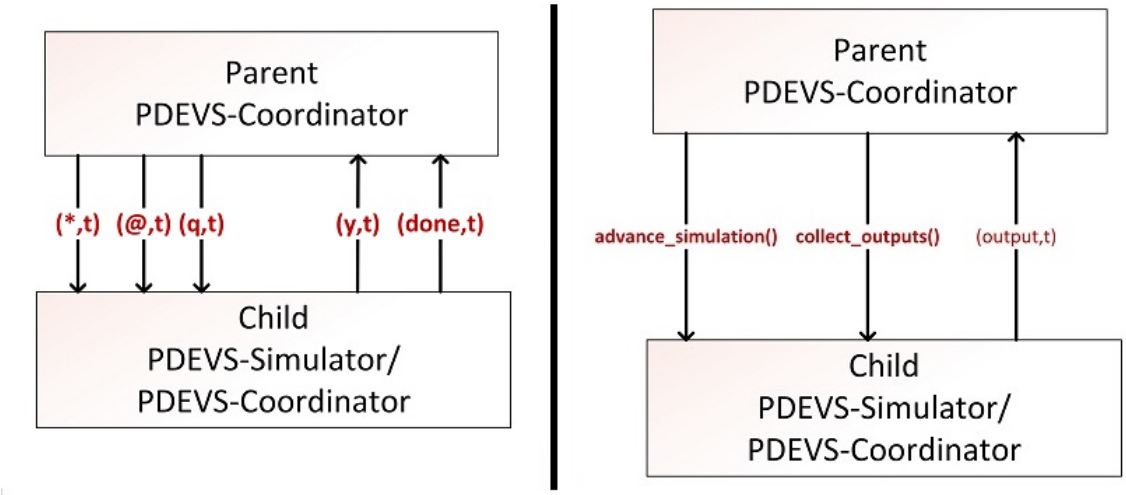

Figure 4.7: Comparison of Communication Mechanisms

We will now discuss the specific algorithms used by the coordinators and simulators in the new communication mechanism.

Coordinator Algorithms

Nodes in the processor (simulator/coordinator) hierarchy tree communicate via function calls and replies. Because, multiple message types are supported, replies can carry more than the confirmation of execution. The returned message is encoded as to support different kind of messages (both done and input/output messages). Afterwards, the message type is processed and then routed accordingly. In embedded CDBoost a particular message structure has been defined and is used to include port structures used for communication with hardware components.

The following listing (Listing 4.1) shows what takes place when the "*collect_outputs*" function is invoked on a coordinator. The coordinator verifies if it has reached its next state change time. If not, an empty reply is sent; otherwise, the outputs of each imminent coordinator associated with a submodel in the EOC

(models that send signal to the outside/environment) are collected and added to the output bag.

Coordinator
Vars:
next // Next scheduled event time
last // Last processed event time
FEL // Future event list

```
Method collect_outputs(Time t)
        if  t != next then
                return {}
        else
                set outputs = empty bag
                for each imminent submodel Coordinator c
                        if c is in EOC then
                                outputs = Union(outputs, c.collect_outputs(t))
                        end if
                end for
                return outputs
        end if
end method
```

Listing 4.1: Coordinator - collect_outputs() [9]

For *advance_simulation*, the time t is verified to ensure that it is between the last and next scheduled change. If so, t is saved as the last change time, and external imminent models (those that received events from the environment) are set by adding each receiver of the external coupling set to the external imminent set, and adding the content of the inbox to the receiver's inbox. The previous steps run if the coordinator inbox is not empty (i.e. an input message was received) and the receiver's next state change is not t. If it is time for the next state change ($t == next$), the outputs of each imminent model are collected and carried out to any linked coupled model (internally coupled) that is then added to the external imminent set. Note that this set refers to models that become "ready to execute" because an external event happen, and not

because of an internal delay expiration.

In all these cases, the coordinator calls *advance_simulation* for each coordinator in the *imminent* and *external_imminent* sets, and their next state change time is added to the Future Event List (FEL). If the FEL is empty, the next state change is infinity; otherwise, it is picked from the FEL. Finally, all the *imminents* are retrieved from the FEL.

```
Method advance_simulation(Time t)
        assert t in [last, next]
        last = t
        set external_imminents = empty set

        for each Coordinator c of a submodel in EIC
                if self.inbox is not empty and c.next != t then
                        add r to external_imminents
                end if
                add self.inbox contents to c.inbox
        end for

        if t == next then
                for each Coordinator c of a submodel
                receiving input from an imminent i because of IC
                        set temp = i.collect_outputs()
                        if not empty temp and c.next != t then
                                add c to external_imminents
                        end if
                        add temp to c.inbox
                end for
        end if

        for each Coordinator c in Union(imminents, external_imminents)
                c.advance_simulation(t)
                if c.next != infinity then
                        FEL.remove_value(c) //for rescheduling
                        FEL.insert(c.next, c)
                end if
        end for
        if empty FEL then
                next = infinity
        else
                next = FEL.top.first
```

```
        end if
        imminents = coordinators on top of FEL
        remove imminents from FEL
end method
```

Listing 4.2: Coordinator - advance_simulation() [9]

Simulator Algorithms

For simulators - leaf nodes in the processor hierarchy tree - only the return reply mechanism is required. Since coordinators and simulators do not know their parents, all communications are initiated from top to bottom, and the replies are collected using the method returned values. The function names are the same as in the Coordinator: *advance_simulation* and *collect_outputs.* The algorithms for the Simulator are shown in the next listings.

The *collect_outputs* method verifies the time parameter *t*. If it not time yet for the next scheduled event, an empty bag is returned; otherwise, the output generated by the model is returned.

Simulator : subclass of Coordinator
Vars:
next // Next scheduled event time
last // Last processed event time
model // atomic model being simulated

```
Method collect_outputs(Time t)
        if  t != next then
                return {}
        else
                return model.out()
        end if
end method
```

Listing 4.3: Simulator - collect_outputs() [9]

For *advance_simulation*, we verify if time t is valid by making sure it is within the last change and the next expected change. If *advance_ simulation* was called with a legitimate time t, the *inbox* content is checked. If the inbox is empty and it is time for the next event, i.e. the next internal transition, the internal function is executed and the next change is set by adding the last change time and the delay TA. When *inbox* is not empty (i.e. an input has been received), we execute the external function if the time different from the next state change (internal transition time). If not, it indicates that the external and internal transitions are scheduled for the same time and the confluent function is therefore executed.

```
Method advance_simulation(Time t)
        assert t in [last,next]
        if self.inbox is empty and t == self.next then
                model.internal()
                next = last + model.time_advance()
        end if

        if self.inbox is not empty then
                set local_t = t - last
                if t == next then
                        model.confluence(inbox, local_t)
                else
                        model.external(inbox, local_t)
                end if
                next = last + model.time_advance()
        end if
        last = t
end method
```

Listing 4.4: Simulator - advance_simulation() [9]

These algorithms therefore implement the DEVS execution mechanism provided in the abstract simulator by using a different communication mechanism. In the following section, we will see how the modeling subsystem is designed to allow the user to easily define models.

4.3.3 Modeling Subsystem

The user implements atomic and coupled models by extending two basic classes: *PDEVSAtomic* and *PDEVSCoupled*. Figure 4.8 illustrates model classes that form the modeling subsystem in Embedded CDBoost. We will first present the *PDEVSAtomic* class, which is used to define new atomic models. The constructor requires two template parameters: Time and Message. Virtual functions provided by *PDEVSAtomic* correspond to those described in the formalism: internal, external, confluent, time-advance and output functions. The time advance function, which is commonly included in the internal and external functions in various simulation implementations, is here clearly separated and has its own dedicated function. In addition, an *init* function setups the model initial state.

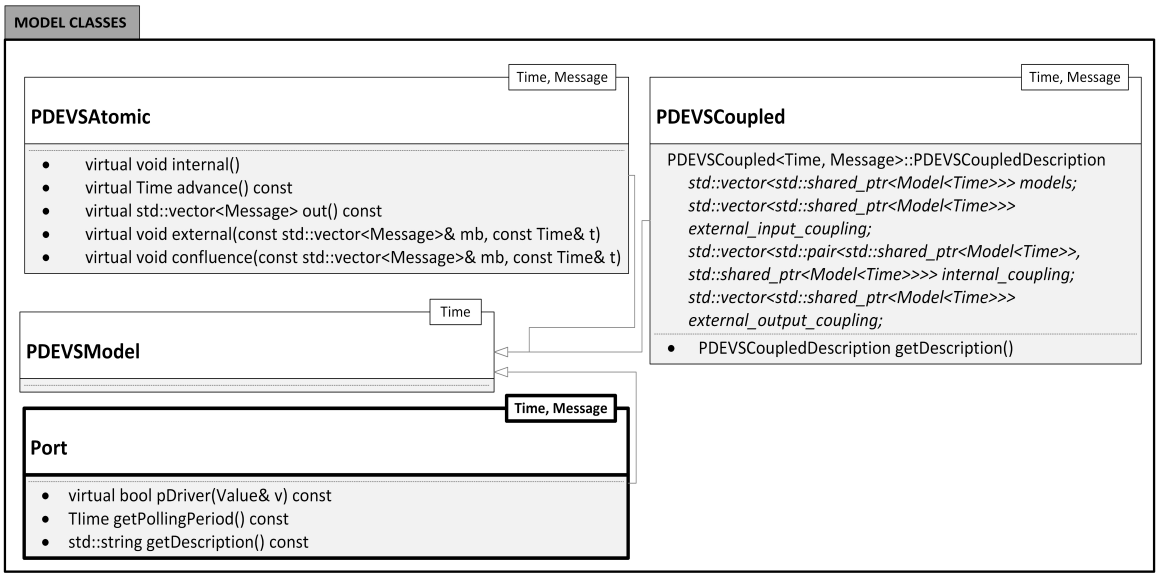

Figure 4.8: Model Classes

Coupled models are defined using the *PDEVSCoupled* class. This class constructor receives four parameters: the list of pointers of the components to be coupled; the External Input Couplings (EIC) pointers list for models that receive inputs from

outside of the coupled model; the Internal Couplings (IC) pointers list for models that are connected internally; and finally the External Output Couplings (EOC) pointers list for models that send outputs outside of the coupled model.

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

PDEVSAtomic and *PDEVSCoupled* are common to both CDBoost and Embedded CDBoost. Embedded CDBoost adds " Port" (shown in bold in Figure 4.8) to its modeling subsystem. To allow communication with hardware components in Embedded CDBoost, the user must provide ports implementation by extending a Port base class, and specifying a " *pDriver*" or port driver function to translate model output values to specific hardware components commands and vice-versa. For an input port, pDriver could provide what GPIO (General Purpose Input Output) to read and set the port value for instance. For an output port, the pDriver function receives a value that is then translated into actions such as writing to a GPIO to turn on an LED.

4.3.4 Execution Subsystem

Execution classes, illustrated in Figure 4.9, implement the abstract simulator algorithms and execute models. The *PDEVSCoordinator* class, in charge of managing coupled models, requires three template parameters: *Time*, *Message* and *Future Event List (FEL)*. These three parameters will be detailed in the ancillary subsystem with *utility* classes.

Contrary to E-CD++ where the processor tree is constructed by the main runtime subsystem, this step is performed upon the invocation of a processor constructor. Constructing coordinator objects is complex, as it requires the coupled model components to be extracted and embedded in the coordinator. For instance, when the

coordinator is built, all the children are constructed, and the couplings between components that communicate are saved. The algorithms described previously (listing 4.1 and 4.2) - *collect_outputs* and *advance_simulation* (renamed *advance_execution* since it used the real time execution time) - are implemented in these classes.

The *PDEVSSimulator* class implements the simulator's algorithms presented in listing 4.3 and 4.4. Therefore, this class is in charge of calling the state transition functions at the appropriate times, and of returning the outputs of the atomic models to their coordinators.

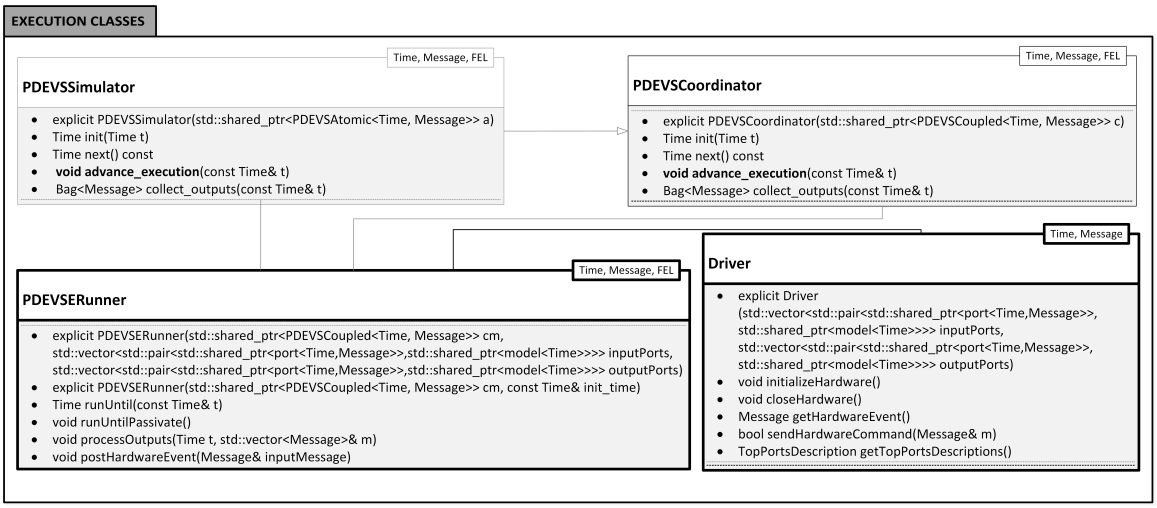

Figure 4.9: Execution Classes

Embedded CDBoost also uses the flattened coordinator technique and adds a global driver ("Driver" shown in bold) that manages ports like in E-CD++. The Driver is responsible of initializing hardware, retrieving inputs from hardware components connected to input ports (by calling the appropriate port pDriver method), and sending commands to hardware components connected to the output ports. The input event retrieving mechanism is based on the same polling interrupt driven mechanism and communicates with the *PDEVSERunner*. This class is associated with the top or root coordinator, manages the global execution and defines the end time of

the simulation. It also provides mechanisms for output and debugging.

Runner (Root Coordinator) Algorithm

The runner used in Embedded CDBoost is particularly different from CDBoost since input have to be gathered from the hardware and output sent to the target platform too.

```
run():

while curentTime < stopTime /* main loop - this can be forever */
        wait for is signals from environment or internal time out(tN)
        if external event then
                Message in = DX(is)
                topCoordinator.postEvent(in) //add in to top coordinator inbox
                topCoordinator.advance_execution()
        else if internal time out then
                topCoordinator.collect_outputs()
                if output messages out received then // process outputs
                        os = DY(out)
                        send os signal to hardware
                end if
                topCoordinator.advance_execution()
        end if
        tN = topCoordinator.next()
end while
```

Listing 4.5: eRunner algorithm

The runner executes the application for the time specified by the user. The default stop time is infinity as in typical embedded systems where a program is set to run repeatedly forever. It waits for an internal time out to happen or an external event in order to advance model execution. In the first case, outputs are collected and *advance_execution* called with the current time. When an external event occurs, the event value is added in the top coordinator and *advance_execution* called in order to process the event. When the runner receives an output message, it is processed and a corresponding value is sent to the concerned port.

4.3.5 Ancillary Subsystem

The *utility* classes (Figure 4.6) provide essential data structures to run the execution properly. The first class in the utility category is called *Message*. Boost::any is used by default in CDBoost as the message type, and it allows the exchange of any type of messages in our models. In E-CDBoost, we have defined a special message type that includes time, port, and value parameters.

For the *Time* component, it is associated with the physical time in Embedded CDBoost and provides a real-time clock with microsecond precision. It is interfaced with a 32-bit hardware timer.

The Future Event List (FEL) is also provided as part of the utility classes. Using an effective FEL is essential in order to achieve good performance. For the FEL type, any structure that matches the priority queue signature is accepted. Hence, the user can define customized schedulers and increase performance if needed. The default provided FEL is the standard priority queue. This data structure is part of the language and is suited to store and retrieve timed events.

In addition to the data types provided by the above-described classes, an input stream model is also provided. Its role is to allow reading and processing events that originate from an external source in simulated mode.

4.3.6 Execution on the Target Platform

Embedded CDBoost runs on top of the microkernel presented in section 4.1. This latter handles its system calls and provides requested services to the real-time executive. The real-time executive communicates with hardware mainly via ports and drivers using a similar mechanism to the one presented for E-CD++. The same hardware abstraction layer is used here in order to streamline the development and

ease applications porting. The hardware abstraction layer and hardware platforms will be presented in the next chapter. Chapter 6 will illustrate concrete examples of applications developed using the two real-time executives.

Chapter 5

Hardware Related Layers

Because hardware goes hand in hand with software in embedded systems, this chapter is dedicated to the hardware-dependent kernel layer and the hardware layer of our bare-metal solution.

5.1 Hardware Peripheral Libraries Integration

Recall from section 2.5 that MCU vendors provide device libraries to allow access and configuration of hardware components, and ease the software development task. We integrate these peripheral libraries in the hardware-dependent kernel layer as illustrated in figure 5.1 (The Hardware Peripheral Libraries layer). They are used by the HAL layer and by the user defined port pDriver methods if needed.

When defining input and output ports connected to hardware, the user can reuse methods provided by the hardware abstract library (The HAL layer shown in figure 5.1) to get value from sensors or actuate motors for instance. The HAL layer contains ready-to-use methods in order to include commonly used hardware components (e.g. ultrasonic, light sensors) easily and test the resulting application. This is particularly useful since writing hardware components/devices drivers from scratch can be

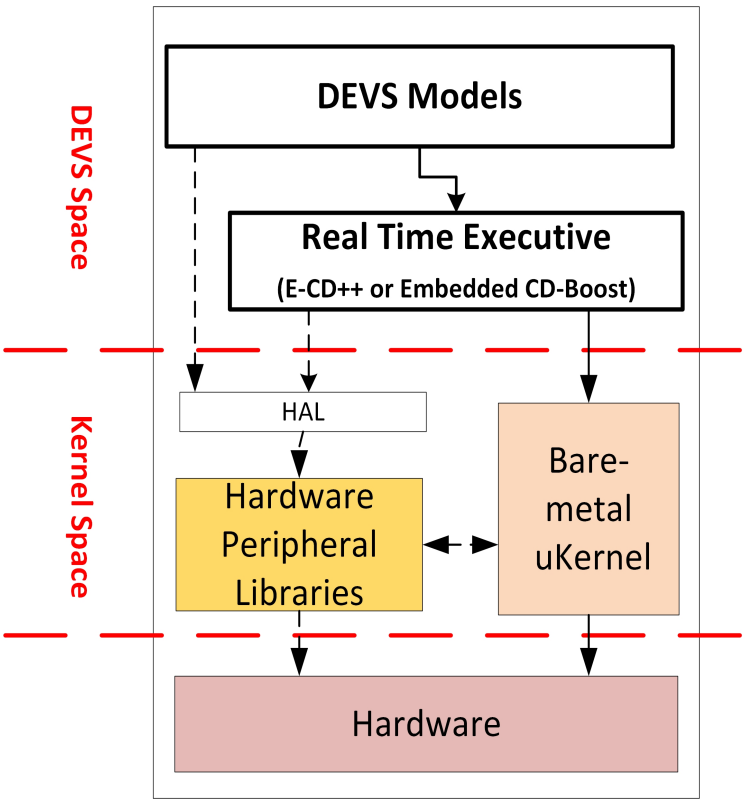

Figure 5.1: Overview of the Bare-Metal Design

particularly time-consuming. The HAL layer resides on top of the vendor provided peripheral libraries.

At the beginning, since most of the development boards (See Appendix B) we used were from STMicroelectronics, we included their standard peripheral libraries, and built a HAL layer that provides function related to ultrasonic sensors (such as the HC-SR04), light sensor (QTR-1RC), servomotors. However, for every new component, the user needs to be familiar with STM peripheral libraries and understand the electronic features of the component in order to write its drivers. This can be particularly challenging for beginners or developer who only have a software background.

Later, with the introduction of STM32Cube (presented in section 2.5 as well), we integrated the possibility of using the STM32Cube graphical interface in order to initialize and configure peripherals, and include the generated code into the project.

For this to happen, each project is structured in four main components: internal (contains the real-time executive files), user_models (contains the models the user defines), user_drivers (ports and files related to hardware components interfacing), and finally hal_libraries (vendor specific peripheral libraries). It is in this later that the code autmatically generated from STM32Cube can be included, and reused in ports. The Eclipse IDE that we use includes a STM32Cube plug-in to enable this.

Although integrating the STM32Cube option helped to reduce the steep learning curve associated with vendor-dependent peripheral libraries, we wanted a more general solution that was vendor independent and provided a friendlier API. We therefore decided to include MBED internal libraries as the default hardware peripheral layer.

5.2 MBED Integration

We have included the MBED internal libraries as part of the bare-metal layers in order to allow the modeler to easily write port driver functions and enable fast prototyping. The MBED API is only invoked in the port pDriver functions and provides access to hardware peripherals in order to sense or actuate components. Integrating both DEVS models and the MBED API greatly improves the embedded software development process, provides support for numerous devices, and reduces development time and cost.

In the HAL layer, MBED libraries for common components and shields such as the Parallax Robot Shield were developed and deployed for the MBED community to use. For this Parallax robot library we defined functions that allows to spin onboard servomotors at a desired speed and direction. Whiskers (integrated touch sensors that comes with the robot) and additional sensors (ultrasonic, bump, and reflectance) access methods were included in the library, and sample navigation programs provided.

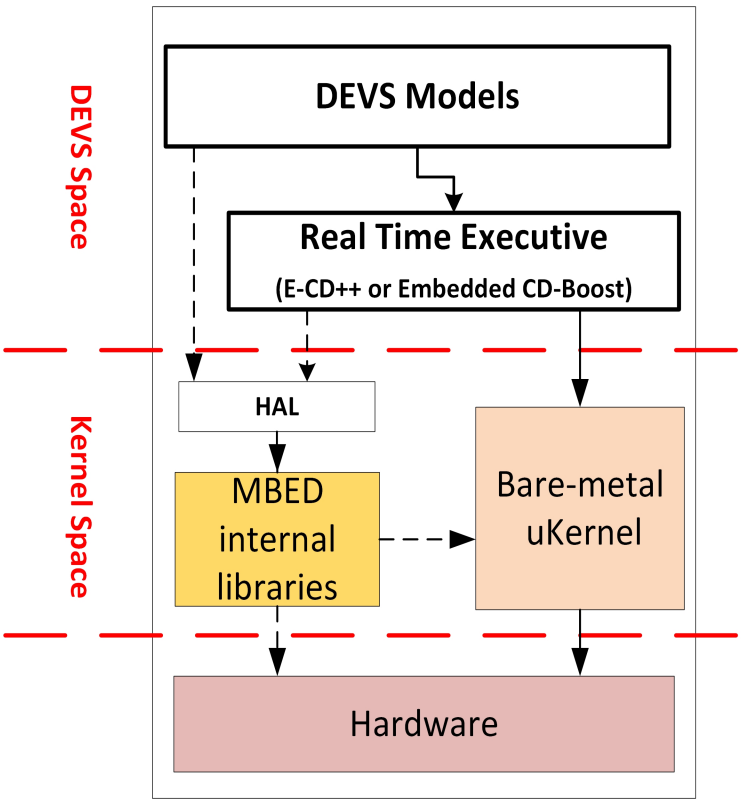

Figure 5.2: Overview of the Bare-Metal Design with MBED

This helps to reduce the development effort required for device drivers and will allow future users to directly use the Parallax robot shield library in the port definitions.

Besides, it is easy to develop new device drivers with the MBED friendly object oriented API. The learning curve is considerably faster. Plus, multiple device libraries are readily available and can be reused by the user. For instance, some other robotic platforms such as the popular Pololu M3PI mobile platform exist, as well as libraries for connectivity shields (e.g. Bluetooth, WiFi and NFC) are available. Moreover, the developed drivers using the MBED API can be reused across different vendor platforms since the API is vendor-independent. Only the MCU dependent layers need to be updated.

The other project that is currently under development is a DEVS library that can

be used in the online MBED compiler and accessible to the MBED developer community. It would be the first model-based library, in our knowledge, to be available for deployment onto MBED platforms.

5.3 Summary

To recapitulate the bare-metal solution design, we have DEVS models that are provided by the modeler, as the first layer. These DEVS models are executed by a real-time executive, i.e. the new E-CD++ or Embedded CD-Boost. These first two layers are built on DEVS theory and thus enclosed in a DEVS space. The real-time executive uses services provided by the bare-metal microkernel for actions such as file reading. It also invokes a peripheral driver library (i.e. MBED, or a vendor specific library) depending on the modeler library choice. The default provided library being MBED since it is vendor independent and ideal for fast prototyping.

Chapter 6

Applying DEMES - A Case Study

Recall from chapter 2 that model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing. We have followed the DEMES approach, presented in section 2.2 to build several applications and executed them on the target platform using the new kernels. In this chapter, we will particularly focus on one application and present how it evolved progressively from its system of interest definition, to formal model, to the real system.

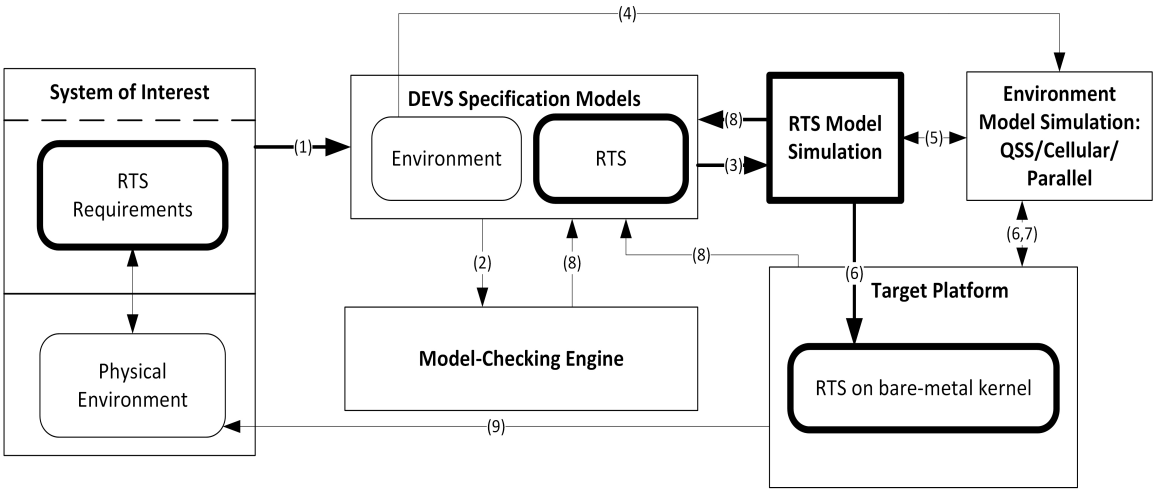

Figure 6.1: DEMES-based Development Cycle

Figure 6.1 outlines in bold the DEMES development phases we will focus on for

this particular case study. We will start by defining the system of interest, specify the corresponding DEVS models, then explore different test scenarios using simulation, and finally emphasize on DEVS models execution on the new bare-metal kernels.

We have used this approach to build several applications, many of these are related to autonomous vehicles and include a robocart and a self-driving vehicle. The robocart navigates using an ultrasonic and obstacle sensor to avoid obstacles and find a free path. The self-driving car was designed to follow road marks such as line and respect other vehicles driving in the same line. Four light sensors were used to keep track of the road lines, and an ultrasonic sensor to detect vehicles and evaluate the distance between. We are currently running different tests for this system. Other non-robotic related applications include a simple parking assist device that emits a green, yellow, and red light to ease reverse parking maneuvers. We are also working on a drone controller and intend to develop biomedical and IoT related applications soon. In this section, we will focus on a line tracking robot application and its DEMES-based development cycle.

6.1 System of Interest

The definition of a system of interest is the first step in the DEMES-based development cycle. It involves two parts: the real-time system requirements and the physical environment identification. We will particularly focus on the desired system: a line-tracking robot. It is an autonomous robot capable of following a line and able to take decisions to get back on track. The requirements are as follows: the robot shall be equipped with a light sensor that faces the ground and measures the amount of light reflected off a small ground surface. The controller should consider a medium

percentage of reflected light as a detected path and initiate the robot to move forward. When the robot goes off track, i.e. does not sense a path trail; it stops, turns counter-clockwise slightly, and then tries to detect a trail again. If a path is detected, the robot moves forward again; otherwise, it continues to turn until it finds a path to follow. The robot should also be able to receive manual signals to start and stop.

For the physical environment, we will use an existing path designed for the Lego Mindstorms robot and previously used to test the same application (running on the Xenomai-based E-CD++ ([111]).

6.2 DEVS Model Specification

Once the system of interest is defined, the following step is to model the system using DEVS. This formalism, as introduced in section 2.2, decomposes complex system designs into basic/behavioral models (atomic models) and composite/structural models (coupled models). We take a top down approach and first define the structure of the line tracking robot system. Multiple iterations are usually required to capture the requirements into an appropriate hierarchical structure. Figure 6.2 [111] illustrates the resulting hierarchy for our example.

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

In terms of components, the sensor unit contains sensors or input devices. In this case, it contains an atomic model, the light sensor, which reads the amount of light

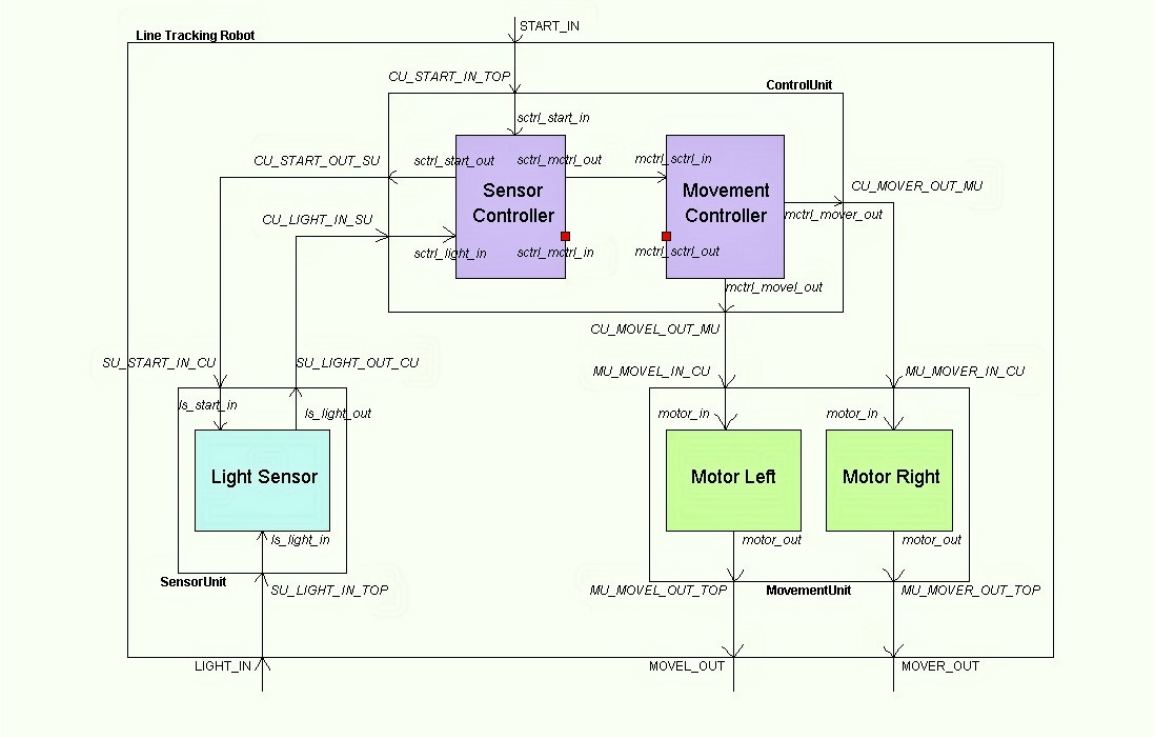

Figure 6.2: Line Tracking Robot Model Hierarchy Diagram

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

the signals they receive from the control unit.

In addition to the above textual description of each coupled and atomic models, DEVS models can be formally specified using the notation presented in section 2.3. These formal specifications are the basis of model-checking, or formal verification basis from which a modeler, or an automated tool can verify system properties analytically. The initial DEVS model specification is also preserved as much as possible throughout the development cycle. The control unit coupled model specification and the sensor controller specification are available in Appendix C.

The DEVS specification in Appendix C is particularly useful to verify analytically general system properties. A graphical form can also be used to illustrate better the atomic model behavior and integrate all its state transition and output functions. Figure 6.3 illustrates the DEVS Graph representing the sensor controller's behavior. The state diagram summarizes the behavior of a DEVS atomic component by presenting the states, transitions, inputs, outputs and state durations graphically. The circles represent states and the double circle is the initial state. The name and duration of a state is shown in the circle. The continuous edges between the states represent the external transitions, which includes the names of the input ports, the input value and any condition on the input (with format "port?value"). The dotted lines represent the internal transitions and the associated outputs (with format "port!value").

The Sensor Controller starts in the IDLE state and remains in that state until a start command is received. Once issued, an external transition is triggered and the Sensor Controller state changes to PREP_RX. At this stage, it waits for a defined time $ta = scRxPrepTime$ after which a "start" output signal is sent to the Light Sensor and an internal transition is triggered changing its state to WAIT_DATA. The Sensor Controller waits in this state until it receives a signal from the Light Sensor. When

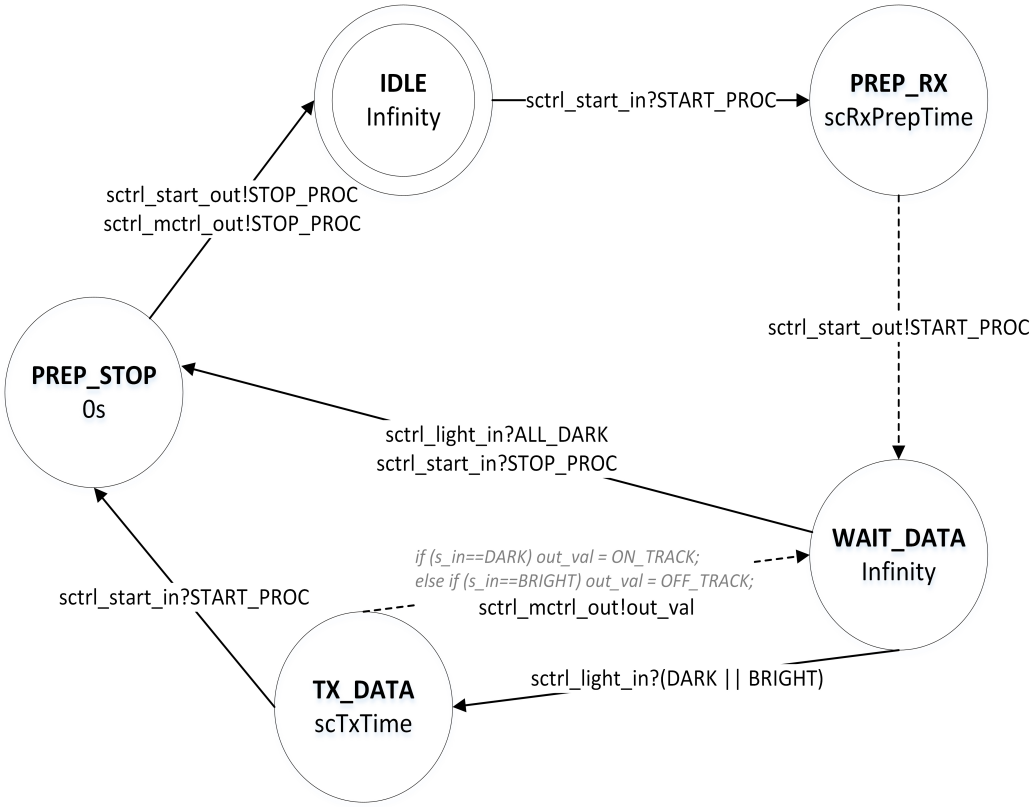

Figure 6.3: Sensor Controller State Diagram

a signal is received, if the signal indicates that the robot reached the destination (signal value is ALL_DARK), an external transition causes the Sensor Controller to switch to the PREP_STOP state, at which it will immediately send a stop signal to the Light Sensor and the Movement Controller and then transition back to the IDLE state. However, if the received signal is different, the Sensor Controller will go to the TX_DATA state at which it will wait for a time advance period of $ta{=}scTxTime$ before sending an output signal to the Movement Controller indicating whether the robot is on track or not, and going back to the WAIT_DATA state. At any point in time, if the Sensor Controller receives a manual stop signal (STOP_PROC), it will execute an external transition to the PREP_STOP state to stop all activities.

The previous DEVS representation can be transformed into a Timed Automaton

[82] and verified using model-checking tools such as UPPAAL. This out of scope for this thesis but further details about such transformation and model checking can be read in [82] and [85]. Depending on the outcome of model checking (such as deadlocks, liveness problems, unbounded time), adjustments can be made to the original specification and verification run again. The formal proofs obtained in this step can be complemented with simulation where individual behaviors of sub models can be tested under specific conditions.

6.3 RTS Model Simulation - Virtual Environment Testing

RTS Model simulation allows the user to test the defined models in a virtual environment and under various settings. To run simulations and test the system components in multiple scenarios, we can use non-real time (CD++ and CDBoost) or their real-time counterpart tools (E-CD++ and Embedded CDBoost) that offer real-time and hardware-in-the loop capabilities. We will focus on these latter but first show how CD++ and E-CD++ can be used together to go from simulated model to executable model.

CD++ was developed to run in a simulated environment and only provides simulated results. To allow execution on the target platform, E-CD++ has to be used. As mentioned in section 4.2, E-CD++ was designed to be compatible with CD++. Therefore, the models simulated in CD++ can be loaded as is in E-CD++ for embedded execution.

In CD++, The user defines atomic models, identified during the DEVS model specification phase, as subclasses derived from the model class that specify the internal, external and output transition functions. The model hierarchy is specified in a

model file, and contains model components and their port links. The model file (also called MA file) of the line tracking robot is shown below.

```
[top]
components : ControlUnit MovementUnit SensorUnit
in : LIGHT_IN
in : START_IN
out : MOVER_OUT
out : MOVEL_OUT
%input connections
Link : LIGHT_IN SU_LIGHT_IN_TOP@SensorUnit
Link : START_IN CU_START_IN_TOP@ControlUnit
%output connections
Link : MU_MOVER_OUT_TOP@MovementUnit MOVER_OUT
Link : MU_MOVEL_OUT_TOP@MovementUnit MOVEL_OUT
%internal connections
Link : SU_LIGHT_OUT_CU@SensorUnit CU_LIGHT_IN_SU@ControlUnit
Link : CU_START_OUT_SU@ControlUnit SU_START_IN_CU@SensorUnit
Link : CU_MOVER_OUT_MU@ControlUnit MU_MOVER_IN_CU@MovementUnit
Link : CU_MOVEL_OUT_MU@ControlUnit MU_MOVEL_IN_CU@MovementUnit

[SensorUnit]
components : LS@LightSensor
in : SU_LIGHT_IN_TOP
in : SU_START_IN_CU
out : SU_LIGHT_OUT_CU
%input connections
Link : SU_LIGHT_IN_TOP ls_light_in@LS
Link : SU_START_IN_CU ls_start_in@LS
%output connections
Link : ls_light_out@LS SU_LIGHT_OUT_CU

[ControlUnit]
components : SCtrl@SensorController MCtrl@MovementController
in : CU_START_IN_TOP
in : CU_LIGHT_IN_SU
out : CU_START_OUT_SU
out : CU_MOVER_OUT_MU
out : CU_MOVEL_OUT_MU
%input connections
Link : CU_START_IN_TOP sctrl_start_in@SCtrl
Link : CU_LIGHT_IN_SU sctrl_light_in@SCtrl
%output connections
Link : sctrl_start_out@SCtrl CU_START_OUT_SU
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
%currently unused connection
%Link : mctrl_sctrl_out@MCtrl sctrl_mctrl_in@SCtrl
```

```
[MovementUnit]
components : MotorR@MotorRight MotorL@MotorLeft
in : MU_MOVER_IN_CU
in : MU_MOVEL_IN_CU
out : MU_MOVER_OUT_TOP
out : MU_MOVEL_OUT_TOP
%input connections
Link : MU_MOVER_IN_CU motor_in@MotorR
Link : MU_MOVEL_IN_CU motor_in@MotorL
%output connections
Link : motor_out@MotorR MU_MOVER_OUT_TOP
Link : motor_out@MotorL MU_MOVEL_OUT_TOP

[LS]
lsTxTime : 00:00:00:050

[SCtrl]
scTxTime : 00:00:00:050
scRxPrepTime : 00:00:00:050

[MCtrl]
movePrepTime : 00:00:00:050
turnTime : 00:00:01:000

[MotorR]
movePrepTime : 00:00:00:050

[MotorL]
movePrepTime : 00:00:00:050
```

Listing 6.1: Line Tracking Robot Model File

The model file describes the structure illustrated in figure 6.2. It starts by specifying the main components, i.e. the sensor unit, the control unit and the movement unit, described top ports and connections between the components. Each non atomic component is then described using the same approach. For instance, the control unit portion specifies the two atomic models components: SCtrl (a sensor controller instance) and MCtrl (a movement controller instance). Then, the input (CU_START_IN_TOP and CU_LIGHT_IN_SU) and output (CU_MOVEL_OUT_MU and CU_MOVER_OUT_MU) ports of the Control Unit are defined. Finally, the input and output connections between the ports of the Control Unit and those of SCtrl and

MCtrl are described, as well as the internal connections between SCtrl and MCtrl. The direction of the connection is read as FROM port → TO port.

To show an example of an atomic model implementation, the following code describes the transition and output functions of the *Sensor Controller*. A correspondence can be seen with the state diagram shown earlier in figure 6.3 as well as the DEVS specification it is derived from. The code snippet first shows a portion of the external transition function that describes the transition from state WAIT_DATA to either TX_DATA or PREP_STOP depending on the value (sensor_input) of the incoming signal from the *Light Sensor* received on port sctrl_light_in. Lines 18 to 27 show a portion of the internal transition function describing the transition from TX_DATA to WAIT_DATA. Finally, lines 29 to 44 show a portion of the output function's behaviour at state TX_DATA. The output function sets the output signal (ON_TRACK or OFF_TRACK) to send to the *Movement Controller* through the sctrl_mctrl_out port.

```
1  Model &SensorController::externalFunction( const ExternalMessage &msg ) {
2    //...
3    if (msg.port() == sctrl_light_in){  // Light sensor signal received
4      if(state == WAIT_DATA) { // Sensor controller was waiting for data
5        sensor_input= msg.value();   // Get the light sensor input
6        if(sensor_input==ALL_DARK) { // Destination Reached
7          state=PREP_STOP;           // Prepare to stop immediately
8          holdIn(Atomic::active,ZERO_TIME );
9        } else {                     // Robot is not at destination yet
10         state = TX_DATA;           // Sensor goes into transmitting state
11         holdIn(Atomic::active, scTxTime );// wait for scTxTime
12       }
13     }
14   }
15   return *this;
16 }
17
18 Model &SensorController::internalFunction( const InternalMessage & ) {
19   switch (state){
20      //...
21     case TX_DATA:       // Just transmitted data to movement controller
```

```
22        state = WAIT_DATA;      // Wait for new data from the sensor
23        passivate();            // stay in this state until new event
24        break;
25      }
26    return *this;
27  }
28
29  Model &SensorController::outputFunction( const InternalMessage &msg ) {
30    switch (state){
31      //...
32      case TX_DATA: {                  // In transmitting state
33        int output_val;
34
35        if(sensor_input==DARK)        // Light sensor indicates a dark line
36          output_val = ON_TRACK;    // Output signal to MCtrl is ON_TRACK
37        else if(sensor_input==BRIGHT)// Light sensor reads a bright surface
38          output_val = OFF_TRACK;  // Send off_track signal to MCtrl
39        sendOutput(msg.time(),sctrl_mctrl_out, output_val); // Output to MCtrl
40        break;
41      }
42    };
43    return *this ;
44  }
```

Listing 6.2: Sensor Controller Implementation Snippet

Once the models have been implemented, CD++ can be used to run different simulations on the host workstation, and revise the specified models if needed. It is an iterative and incremental process.

Diverse scenarios can be tested by using event files that injects events through the input ports of the model. To carry out these experiments, an event file that specifies the event time, input port and its value is used. Table 6.1 shows the port (input and output ports) mapping table and the description of each value.

Table 6.1: Port Mapping

Port Name	*Port Value*	*Hardware Command*	*Description*
START_IN	10	START	Manual Start Command
	11	STOP	Manual Stop Command
LIGHT_IN	0	BRIGHT	No line detected
	1	DARK	Line detected
	2	ALL_DARK	Destination Reached
MOVER_OUT/ MOVEL_OUT	0	STOP	Stops the motor
	1	GO_FWD	Spins Clockwise
	2	GO_REV	Spins Anticlockwise

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

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

Listing 6.3: Event File

After 1s, we start the system by sending an input to the START_IN input port. Then, at 2s, a value of 1, indicating a line detection, is sent through the LIGHT_IN input port. To test situations when the robot gets off-track, a value of 0 is sent through the LIGHT_IN port. Sending 11 through the START_IN port then simulates a manual stop. Different values are sent through the LIGHT_IN port to test how the system behaves in a stop state. Afterwards, the system is started again, and bright (0), dark (1) and all dark (2) surfaces are alternately sent through the LIGHT_IN port. ALL_DARK signals that the robot has reached its destination and acts as an

automatic stop signal. More values are sent through the LIGHT_IN port and finally a manual stop signal is sent through the START_IN port.

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

The following table sums up the results obtained with the previous event file in CD++:

Table 6.2: CD++ Simulation Results

Input	Output	Description
1. 00:00:01:000 START_IN START	-	System START - no output
2. 00:00:02:000 LIGHT_IN DARK	00:00:02:200 mover_out GO_FWD	Path detected - motors go forward
	00:00:02:200 movel_out GO_FWD	
3. 00:00:02:500 LIGHT_IN BRIGHT	00:00:02:600 mover_out STOP	OFF Track signal - motors stop
	00:00:02:600 movel_out STOP	
	00:00:02:700 mover_out GO_FWD	Robot Turns motor_right forward
	00:00:02:700 movel_out GO_REV	and motor_left reverse
4. 00:00:02:700 START_IN STOP	00:00:02:700 mover_out STOP	Manual System STOP - Turn interrupted
	00:00:02:700 movel_out STOP	
5. 00:00:03:000 LIGHT_IN DARK	-	Ignored - System STOPPED
6. 00:00:03:500 LIGHT_IN BRIGHT	-	Ignored - System STOPPED
7. 00:00:05:000 START_IN START	-	System START - no output
8. 00:00:05:500 LIGHT_IN BRIGHT	00:00:05:700 mover_out GO_FWD	OFF Track signal - Robot Turns
	00:00:05:700 movel_out GO_REV	
9. 00:00:06:000 LIGHT_IN BRIGHT	-	Ignored - Still Turning
10. 00:00:06:500 LIGHT_IN DARK	-	Ignored - Still Turning
11. –	00:00:06:650 mover_out STOP	Turn complete - motors stop
	00:00:06:650 movel_out STOP	
12. 00:00:07:000 LIGHT_IN DARK	00:00:07:200 mover_out Go_FWD	Path detected -motors go forward
	00:00:07:200 movel_out Go_FWD	
13. 00:00:07:500 LIGHT_IN DARK	-	Redundant - motors still moving forward
14. 00:00:08:000 LIGHT_IN ALL_DARK	00:00:08:050 mover_out STOP	Reached Destination - motors stop
	00:00:08:050 movel_out STOP	
15. 00:00:08:500 LIGHT_IN DARK	-	Ignored - STOPPED
16. 00:00:09:000 LIGHT_IN DARK	-	Ignored - STOPPED
17. 00:00:09:300 START_IN STOP	00:00:09:300 mover_out STOP	Manual System STOP
	00:00:09:300 movel_out STOP	

We can observe that the described results correspond to the desired behavior. The models can then be moved progressively onto the target platform for validation.

This is done by using E-CD++ that runs on the target platform, provides real-time advance driven by the wall clock time, and allows inputs and output ports to be interfaced with real hardware and environment.

6.4 Execution on the target platform with E-CD++

Transitioning from CD++ to E-CD++ is straightforward. The same model files are supported and do not need to be modified. There is an extra step required for the model file, which is the conversion to hexadecimal as described in section 4.2. This transformation is performed by an internal tool, and it only requires the user to launch the conversion. We injected the same events as in CD++ and compared the obtained results. Inputs are shown as well as their corresponding results (in bold). The format used is <time> <port> <signal_value>. Microseconds are shown in the logs since we used a 32 bit timer that allows such precision. Inputs(numbered lines) and resulting output(unnumbered lines) are shown below.

```
1.    00:00:01:000:023 START_IN START
2.    00:00:02:000:030 LIGHT_IN DARK
      00:00:02:200:119 mover_out  GO_FWD
      00:00:02:200:119 movel_out  GO_FWD
3.    00:00:02:500:021 LIGHT_IN BRIGHT
      00:00:02:600:115 mover_out STOP
      00:00:02:600:115 movel_out  STOP
      00:00:02:700:115 mover_out  GO_FWD
      00:00:02:700:115 movel_out  GO_REV
4.    00:00:02:700:027 START_IN STOP
      00:00:02:700:124 mover_out  STOP
      00:00:02:700:124 movel_out  STOP
5.    00:00:03:000:019 LIGHT_IN DARK
6.    00:00:03:500:030 LIGHT_IN BRIGHT
7.    00:00:05:000:027 START_IN START
8.    00:00:05:500:021 LIGHT_IN BRIGHT
```

```
      00:00:05:700:115 mover_out  GO_FWD
      00:00:05:700:115 movel_out  GO_REV
9.    00:00:06:000:028 LIGHT_IN BRIGHT
10.   00:00:06:500:022 LIGHT_IN DARK
      00:00:06:650:115 mover_out  STOP
      00:00:06:650:115 movel_out  STOP
11.   00:00:07:000:029 LIGHT_IN DARK
      00:00:07:200:122 mover_out  GO_FWD
      00:00:07:200:122 movel_out  GO_FWD
12.   00:00:07:500:031 LIGHT_IN DARK
13.   00:00:08:000:020 LIGHT_IN ALL_DARK
      00:00:08:050:112 mover_out  STOP
      00:00:08:050:112 movel_out  STOP
14.   00:00:08:500:021 LIGHT_IN DARK
15.   00:00:09:000:028 LIGHT_IN DARK
16.   00:00:09:300:027 START_IN STOP
      00:00:09:300:126 mover_out  STOP
      00:00:09:300:126 movel_out  STOP
```

Listing 6.4: E-CD++ Execution Results

The results of this simulation show real-time logs and are identical within a reason (microseconds differences due to kernel differences and the overhead introduced by message processing) to the previous CD++ simulation results, as well as the Xenomai-based E-CD++ version. The target platform was a discovery board in this case. After validating the model behavior on the board, hardware components can be incrementally integrated, and driver interfaces provided to map hardware sensors and actuators to models inputs and outputs.

Simulated components such as the light sensors and motors models are then replaced with real hardware; the control unit model being the one to act as the controller on the target hardware. Three main hardware components are needed for our system: a button to allow the user to send manual start/stop commands, a light sensor, and motors.

We use a QTR-1RC reflectance sensor (infrared LED + phototransistor) [112] as the light sensor, a simple push button and the Parallax robotic shield platform that

includes two servomotors. For the microcontrollers, both Disco and Nucleo boards were used for this application.

In order to interface the model with the previous hardware components, top ports connected to hardware sensors/actuators have to be defined. Recall that ports with drivers enable communication between the interface between the model and hardware sensors/actuators. In this application, we defined four ports - START_IN, LIGHT_IN, MOVEL_OUT and MOVER_OUT - respectively connected to a start/stop button, a light sensor, the left and the right motor. Ports are defined as extensions of a basic port class, and they provide a *pDriver* function that interfaces the input/output message value to a hardware command

The following fragment shows how the START_IN port is defined in E-CD++. The implementation ought to provide a constructor, a pDriver method to interface with its related hardware component, and finally a set of accepted values as per the formal specification.

```
1  class START_IN : public Port
2  {
3  public:
4      START_IN(  const std::string &n = "START_IN", const ModelId &id = 1) : Port(n,
        id){} //Default constructor
5      bool pDriver(double &value);
6
7      enum InputVal{ // Valid input values
8          START,
9          STOP
10     };
11 };
```

Listing 6.5: START_IN Port Definition

pDriver() is shown below; it reads the state (pressed/not pressed) of a button attached to the pin "PC_13" of the board and returns the appropriate input value. Note that functions such as read() are provided by the MBED API. The code required in other cases (e.g. vendor specific peripheral libraries) is much longer and less intuitive.

```
1  // Defined previously
2  DigitalIn user_button(PC_13,PullUp); // Creates button object and setup internal
       pull up resistor
3  //...
4  bool START_IN::pDriver(Value &value){
5
6      if(user_button.read()){ // Button not pressed (read() returns 1 if not pressed
       )
7          return false;
8      }
9      else{
10         if(!isStarted){ // The system is not started yet
11             value = InputVal::START; // 10
12         }
13         else{
14             value = InputVal::STOP;  // 11
15         }
16         isStarted = 1^isStarted; // Started status change
17         return true;
18     }
19 }
```

Listing 6.6: START_IN pDriver()

The value returned by pDriver() is then used by the Driver object to construct an input message sent to the root coordinator as explained in section 4.2.2. Note that in this case the message is only constructed is a button pressed event is reported, i.e. pDriver returns true.

The LIGHT_IN class declaration is similar to START_IN. For the pDriver function, a value corresponding to the light sensor readings is always returned. This exemplifies the difference between active and passive devices. The latter usually return true.

```
1  bool LIGHT_IN::pDriver(Value &value){
2      value = Parallax.lightSensor(D6);
3      return true;
4  }
```

Listing 6.7: LIGHT_IN Port Definition

In this case, the light sensor is attached to the GPIO pin “D6”, the second line calls a function included in a Parallax library that we developed. This library is reusable

and provides different methods to get the values of sensors or actuators present on the Parallax Robot Shield.

MOVEL_OUT and MOVER_OUT are output ports related to the motors (servomotors for the Parallax robotic shield). Their definition is similar to the previous ports except they take an output value and translate it into a hardware command. The following snippet shows the MOVEL_OUT pDriver function. The right motor port driver is similar to the left one except it applies to right servomotor.

```
1  bool MOVEL_OUT::pDriver(Value &value){
2      switch((int) value){
3          case OutputVal::STOP :       // 0
4              Parallax.left_servo(0);  //Stop the left motor
5              break;
6          case OutputVal::GO_FWD :        // 1
7              Parallax.left_servo(20); //FWD the left motor
8              break;
9          case OutputVal::GO_REV :         // 2
10             Parallax.left_servo(-20); //REV the left motor
11             break;
12         default:
13             Parallax.stopAll();       //ERROR - Stop both motors
14             break;
15     }
16     return true;
17 }
```

Listing 6.8: MOVEL_OUT pDriver()

Here again, we use a function defined in our Parallax library. A closer look reveals that this function assigns a specific pulse width to the PWM (Pulse Width Modulated) port associated to the left servomotor in this case.

```
1  void ParallaxRobotShield::left_servo(int speed)
2  {
3      if(speed > 0)                // Clockwise
4          leftServo.pulsewidth_us(leftCenter - (speed * leftClockwiseOffset));
5      else                         // Anti-Clockwise
6          leftServo.pulsewidth_us(leftCenter - (speed * leftAntiClockwiseOffset));
7  }
```

Listing 6.9: ParallaxRobotShield::left_servo()

The offset values are defined during calibration of the servomotors and may vary from one servomotor to another. "pulsewidth_us" is a method provided by MBED and that applies to pulse width modulated pins. Note that we can also use DEVS to pass directly a specific pulse width value for a finer actuator control.

The above defined ports are interfaced with the control unit that acts based on the input values of the button and light sensor, and sends commands to the two servomotors enabling the robot to move forward, turn or stop.

We recorded the resulting execution on the target platform and the exchanged messages that show how the real-time executive works during this process. The listing below shows the messages exchanged between processors, according to the PDEVS abstract algorithm presented in section 2.4, when the user sends a start command at time 00:00:28:191:616.

```
MSG: X / 00:00:28:191:616 / Root(00) / start_in /
10.00000 TO flattop(01)

MSG: * / 00:00:28:191:631 / Root(00) /      0.00000 TO flattop(01)

MSG: X / 00:00:28:191:616 / flattop(01) / sctrl_start_in /
10.00000 TO sctrl(02)

MSG: * / 00:00:28:191:631 / flattop(01) /      0.00000 TO sctrl(02)

MSG: D / 00:00:28:191:631 / sctrl(02) / 00:00:00:500:000 /
0.00000 TO flattop(01)

MSG: D / 00:00:28:191:631 / flattop(01) / 00:00:00:500:000 /
0.00000 TO Root(00)

MSG: @ / 00:00:28:691:631 / Root(00) TO flattop(01)

MSG: @ / 00:00:28:691:631 / flattop(01) TO sctrl(02)

MSG: Y / 00:00:28:691:631 / sctrl(02) / sctrl_start_out /
10.00000 TO flattop(01)

MSG: D / 00:00:28:691:631 / sctrl(02) / ... /      0.00000 TO flattop(01)
```

```
MSG: D / 00:00:28:691:631 / flattop(01) / 00:00:00:000:000 /
0.00000 TO Root(00)

MSG: * / 00:00:28:691:631 / Root(00) /      0.00000 TO flattop(01)

MSG: * / 00:00:28:691:631 / flattop(01) /      0.00000 TO sctrl(02)

MSG: D / 00:00:28:691:631 / sctrl(02) / ... /      0.00000 TO flattop(01)

MSG: D / 00:00:28:691:631 / flattop(01) / ... /      0.00000 TO Root(00)
```

Listing 6.10: E-CD++ - Execution Messages (Start command)

The root coordinator sends a X message - constructed by the global driver object upon getting the value returned by the START_IN pDriver method - followed by a * message These two messages are sent to the flat coordinator (see *flattop* in the trace log) and then directly sent to the sensor controller atomic model (identified by *sctrl* in the trace log). The sensor controller replies with a done message indicating the next change time, i.e. *scRxPrepTime* equals 500 ms in this case. After that time (internal transition expiration), an @ message is sent to the *sctrl* atomic model, and this latter replies with a Y message followed by a done message as directed by the PDEVS abstract simulation algorithms.

In the same way, upon reading the value of a light sensor, the driver constructs the resulting input message and the root sends both X and * messages to the flat coordinator that then sends the messages to the sensor controller. The example below shows what happens when the line is detected.

```
1.  MSG: X / 00:00:47:218:696 / Root(00) / light_in /
1.00000 TO flattop(01)

2.  MSG: * / 00:00:47:218:711 / Root(00) /      0.00000 TO flattop(01)

3.  MSG: X / 00:00:47:218:696 / flattop(01) / sctrl_light_in /
1.00000 TO sctrl(02)
```

```
4.  MSG: * / 00:00:47:218:711 / flattop(01) /      0.00000 TO sctrl(02)

5.  MSG: D / 00:00:47:218:711 / sctrl(02) / 00:00:00:500:000 /
0.00000 TO flattop(01)

6.  MSG: D / 00:00:47:218:711 / flattop(01) / 00:00:00:500:000 /
0.00000 TO Root(00)

7.  MSG: @ / 00:00:47:718:711 / Root(00) TO flattop(01)

8.  MSG: @ / 00:00:47:718:711 / flattop(01) TO sctrl(02)

9.  MSG: Y / 00:00:47:718:711 / sctrl(02) / sctrl_mctrl_out /
5.00000 TO flattop(01)

10. MSG: D / 00:00:47:718:711 / sctrl(02) / ... /
0.00000 TO flattop(01)

11. MSG: X / 00:00:47:718:711 / flattop(01) / mctrl_sctrl_in /
5.00000 TO mctrl(03)

12. MSG: D / 00:00:47:718:711 / flattop(01) / 00:00:00:000:000 /
0.00000 TO Root(00)

13. MSG: * / 00:00:47:718:711 / Root(00) /      0.00000 TO flattop(01)

14. MSG: * / 00:00:47:718:711 / flattop(01) /      0.00000 TO sctrl(02)

15. MSG: * / 00:00:47:718:711 / flattop(01) /      0.00000 TO mctrl(03)

16. MSG: D / 00:00:47:718:711 / sctrl(02) / ... /
0.00000 TO flattop(01)

17. MSG: D / 00:00:47:718:711 / mctrl(03) / 00:00:00:500:000 /
0.00000 TO flattop(01)

18. MSG: D / 00:00:47:718:711 / flattop(01) / 00:00:00:500:000 /
0.00000 TO Root(00)

19. MSG: @ / 00:00:48:218:711 / Root(00) TO flattop(01)

20. MSG: @ / 00:00:48:218:711 / flattop(01) TO mctrl(03)

21. MSG: Y / 00:00:48:218:711 / mctrl(03) / mctrl_mover_out /
1.00000 TO flattop(01)
```

```
22. MSG: Y / 00:00:48:218:711 / mctrl(03) / mctrl_movel_out /
1.00000 TO flattop(01)

23. MSG: D / 00:00:48:218:711 / mctrl(03) / ... /
0.00000 TO flattop(01)

24. MSG: Y / 00:00:48:218:711 / flattop(01) / mover_out /
1.00000 TO Root(00)

25. MSG: Y / 00:00:48:218:711 / flattop(01) / movel_out /
1.00000 TO Root(00)

26. MSG: D / 00:00:48:218:711 / flattop(01) / 00:00:00:000:000 /
0.00000 TO Root(00)

27. MSG: * / 00:00:48:218:711 / Root(00) /      0.00000 TO flattop(01)

28. MSG: * / 00:00:48:218:711 / flattop(01) /      0.00000 TO mctrl(03)

29. MSG: D / 00:00:48:218:711 / mctrl(03) / ... /
0.00000 TO flattop(01)

30. MSG: D / 00:00:48:218:711 / flattop(01) / ... /
0.00000 TO Root(00)
```

Listing 6.11: E-CD++ - Execution Messages(On-track signal)

We can observe that on line 9 an output is generated on the *sctrl_mctrl_out* port, indicating to the movement controller that the robot is on track (5 means " ON TRACK" in this context). This results in the movement controller sending go forward outputs to the motors (line 21 and 22). These outputs are then sent to the root coordinator since they are linked to top output ports. The global driver objects then interprets the Y messages received on lines 24 and 25 and calls the left and right motor pDriver methods causing the robot to move forward. The pDriver functions then call the appropriate MBED function to issue the command.

With the Xenomai-based E-CD++, the Lego's NXT++ library was used to interface the models with hardware i.e. the light sensor and motors. Through a C++

API for Lego NXT robot controller, communication can be established over USB and Bluetooth. However, with this approach, the NXT robot had to be connected to the PC via a USB cable and DEVS models weren't compiled to the native NXT byte code [113].

We deployed the same models on different boards. This time, the native code was directly downloaded in memory via ST-LINK, an in-circuit programmer for the STM32 microcontroller families. Models and driver elements were seamlessly integrated and compiled to the native byte code (using the process explained in section 5.2.) resulting in a DEVS-based firmware able to control the peripherals and respond to diverse external stimuli. The image program is deployed on a Parallax robot shield and runs without the need of being tethered as in the Lego case. Videos of execution with the disco (early debug version) [114] and nucleo [115] boards are available. One of the model weaknesses comes from turning only in the counter-clockwise direction to detect the path, and can lead the robot to do long turns before getting on track as showed in [116] and [117].

6.5 Execution on the target platform with E-CDBoost

We presented in section 4.3 the Embedded CDBoost real-time executive. We will implement the same line tracking robot application and exemplify the execution process with Embedded CDBoost. As with CD++, CDBoost can be used to simulate models on a workstation with virtual time advance and environment settings. Models implementation and execution are different from CD++ and E-CD++. We will show how the execution on the target platform using Embedded CDBoost.

We described in section 4.3.3 the modelling subsystem of Embedded CDBoost.

The user implements atomic models by extending a basic model class and providing the state transition and output functions. The code below shows the example of the sensor controller internal, output and time advance functions using Embedded CDBoost.

```
1  void internal() noexcept {
2      switch (_state){
3          case PREP_STOP:
4              _state = IDLE;
5              _next = infinity;
6              break;
7          case PREP_RX:
8          case TX_DATA:
9              _state = WAIT_DATA;
10             _next = infinity;
11             break;
12     }
13 }
14 /**
15  * @brief advance function.
16  * @return Time until next internal event.
17  */
18 TIME advance() const noexcept {
19     return _next;
20 }
21 /**
22  * @brief output function.
23  * @return a bag of output messages depending on the current state
24  */
25 std::vector<MSG> out() const noexcept {
26     //...
27     switch (_state){
28             case PREP_STOP: //Send stop through sctrl_start_out and mctrl
29                 _outputMessage1 = MSG(portName[sctrl_start_out], STOP_PROC);
30                 _outputMessage2 = MSG(portName[sctrl_mctrl_out], STOP_PROC);
31                 return std::vector<MSG>{_outputMessage1, _outputMessage2};
32
33             case PREP_RX: //Send Start through sctrl_start_out
34                 _outputMessage1 = MSG(portName[sctrl_start_out], START_PROC);
35                 return std::vector<MSG>{_outputMessage1};
36
37             case TX_DATA: { //Send on/off track signals sctrl_mctrl_out
38                 int output_val;
39
40                 if(sensor_input == DARK) output_val = ON_TRACK;
41                 else if(sensor_input == BRIGHT) output_val = OFF_TRACK;
42                 _outputMessage1 = MSG(portName[sctrl_mctrl_out], output_val);
43                 return std::vector<MSG>{_outputMessage1};
```

```
44              }
45          };
46          return std::vector<MSG>{}; //Default: send empty output
47  }
```

Listing 6.12: Atomic Model Implementation Snippet

We can see that the implementation provides the state transition and output functions in a concern to preserve the initial DEVS specification during this step. In this way, it is quite similar to E-CD++ except that the time advance is clearly separated and defined in an advance function called by the internal execution algorithms as explained in section 4.3. Note that the message structure is constructed using the port and the value to be sent. This structure is specific to Embedded CDBoost and is not available in CDBoost. The TIME parameter returned by the time advance function is defined using real time units (i.e. hours, minutes, seconds, milliseconds & microseconds), an addition of Embedded CDBoost too. CDBoost uses integers per default for time advance.

To implement coupled models, input, internal and output links have to be provided once all atomic models have been implemented. The following snippet shows how the control unit coupled model is described in Embedded CDBoost.

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

Listing 6.13: Models Definition with E-CDBoost

We use the same components names as in E-CD++ for this example. The sensor controller instance (sctrl at line 2) and movement controller (mctrl at line 3) are the two components of the control unit. The control unit is created on line 5 by respectively providing its components (**{sctrl,mctrl}**), then its EIC (components

getting signal from hardware components here; sctrl is connected to the light sensor and push button here), its IC (sctrl is connected to mctrl internally), and finally its EOC (components sending output signal to hardware: mctrl to the two motors). One of the advantages of this approach is that no file needs to be embedded onto the target platform or converted beforehand. It also offers a lightweight mechanism for specifying links without the need of a complex parser.

To interface models with hardware components, EIC and EOC components are linked to top ports. Providing port classes' implementations and connecting them to the appropriate models achieve this objective.

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

Listing 6.14: Port and eRunner Declaration

Lines 2 and 3 create the two input ports respectively connected to the start button and the light sensor. Line 5 and 6 show the two output ports linked to the motors. Links between ports and the model they are connected to are passed along with the top model to the *erunner* (defined in section 4.3) that executes models on the target platform.

Like with E-CD++, different tests can be run with simulated components and hardware components incrementally added and tested. Once satisfied with the simulation results, real-hardware components and DEVS controller are integrated.

For hardware integration, we use one of the Seeed Shield Bot (described in chapter 5) onboard reflectance sensors as the input for our light readings, a push button on

the nucleo and the two motors of the Seeed Shield Bot to move the robot.

Top ports connected to hardware sensors/actuators have to be specified to interface the model with the previous hardware components. They enable communication between the model and hardware sensors/actuators. These ports are specified as extension of a basic port class. An example with the LIGHT_IN port follows.

```
template<class TIME, class MSG>
class LIGHT_IN : public port<TIME, MSG>
{
public:
    /**
     * @brief light sensor port class
     *
     * @param n Name assigned to the port.
     * @param polling Polling period associated with the port.
     */
     explicit LIGHT_IN(const std::string &n = "light_in", const TIME &polling =
    TIME(0,0,0,200)) noexcept : port<TIME, MSG>(n,polling) {}
     bool pDriver(Value &v) const noexcept;
};
```

Listing 6.15: LIGHT_IN Port Definition

The LIGHT_IN port is derived from the port class and provides a default polling time (200 ms here) when interrupts are not used by the user. In its pDriver implementation (shown below), we call a function of the Seeed Shield Bot MBED library that returns the value of the onboard sensor used to track the line.

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

Listing 6.16: LIGHT_IN pDriver()

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

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

Listing 6.17: Bot Declaration

This shows how easy changing hardware components is easy unlike the previous case where the hardware components had to be supported by *libplayer* or provide a network interface.

In section 4.3, we explained the execution mechanism used by Embedded CD-Boost. We will illustrate this process using trace logs collected during the execution of the line tracing robot application. It illustrates the advance_simulation/execution() and collect_ouputs() function calls explained earlier in section 4.3. The flattened coordinator forwards the function call to the appropriate simulator which in turns returns outputs or calls its state transition functions. Two examples similar to the ones we provided to illustrate the E-CD++ internal execution mechanism are shown below.

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

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

Listing 6.18: Execution with E-CDBoost (Start command)

The listing above shows the sequence that takes place when the user presses for the first time the start button at time 00:00:02:517:459. The driver constructs an input message that triggers the call of the external function of the sensor controller model. An input message indicating a line detection is then sent by the driver and causes the sensor and movement controller external functions to be called. Two outputs are generated, commanding the motors to go forward.

The listing below shows the case corresponding to a manual stop that causes stop commands to be issued to the motors.

```
DRIVER: INPUT MESSAGE
 Time: 00:02:10:403:002
 Port: start_in Value : 11
 - advance_execution()::flattop
 - advance_execution()::sctrl
        model->external() model->advance(): 00:00:00:000:000
 - collect_outputs()::flattop
 - advance_execution()::flattop
 - collect_outputs()::sctrl
        model->out()
```

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

Listing 6.19: Execution with E-CDBoost (Stop command)

Final Deployment

A video showing the result on the target platform is available here [118].

6.6 Metrics Comparison: E-CD++ vs E-CDBoost

Two of the desired outcomes, stated in chapter 3, were the reduction of the kernel footprint as well as a reduced overhead. We measured these metrics and will present the assessment results in this section.

In terms of code size, kernel design decisions, such as the inclusion of the nanolib — an optimized library for microcontrollers —, allowed us to reduced the code size by more than 50%. We also compared the code size of E-CD++ and Embedded CDBoost. The latter is smaller. For the line tracking robot application for instance, the Embedded CDBoost program occupies 131 KB of flash memory and 448 bytes of data memory while the E-CD++ application takes 240 KB of flash memory and 608 bytes of data memory.

In addition to the footprint evaluation, we compared the overhead introduced by each type of execution mechanism. Indeed, as outlined in chapter 3, one of the

goals of introducing Embedded CDBoost was an effective message passing strategy between model processors that would lead to a decreased overhead. We compared the performance of both techniques for this line tracking robot application. We particularly measured the time it takes for an external event to trigger the external function of a model, i.e. the time it takes from the root to the simulator (EXT: Root to Simulator in Table 6.3). We also assessed the time it takes from the external function to return control to the root (EXT: Simulator to Root in Table 6.3). The other aspect that we examined was the output collection, specifically the time it takes from the root collect outputs command to the output function call (OUT: Root to Simulator) and for the outputs to be received by the driver object (OUT: Simulator to Root). The following table summarizes the results.

Table 6.3: Overhead Evaluation

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

We can observe that the overhead was reduced by more than 60% in all cases. In order to take the above measurements, we used a software instrumentation method. For EXT: Root to Simulator for example, we read the value of a hardware timer when an external event (e.g. new reflected light value) is detected. We then read the value of the hardware timer when the simulator calls the external function of the model (e.g. external() of Sensor Controller), deduct the elapsed time in microseconds and print it using an onboard tracing mechanism, the Serial Wire Viewer more specifically. Although this latter introduces minimal overhead (≈25us) during the measurement process and is recommended for software tracing, we are aware that the using software

instrumentation is not optimal. A hardware instrumentation method (e.g. Toggling a GPIO pin and using a logic analyzer to measure the elapsed time) would certainly be less invasive and give measurements that are more accurate. We would expect the improvements to be in the same proportion in this case too.

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

Another set of tests, not related to this application and that would prove useful, is the case where multiple events are received in a short period of time. This is because CDBoost - the non-real time simulation software version - has proved very effective and achieved results comparable and sometimes better than the fastest DEVS simulator [9].

6.7 Moving from one target platform to another: The MBED advantage

In chapter 3, we also targeted portability and easy porting to new platforms as goals of this project. Using a wrapper around the MBED library allowed us to cover multiple microcontrollers and also preserve the integrity of the initial models; only MCU dependent layers need to be changed when moving from one platform to the other. We ported the line tracking robot application onto a non-STM microcontroller, the Freedom FRDM-64K board, to illustrate that the designed solution is MCU vendor independent and requires minimal porting effort. The element that needs to be updated in this case is mainly the target hardware in the project configuration in order for the new HAL libraries to be included in the project. If pins use a similar standard (for instance both follow the Arduino standard), the user application code and drivers may stay as is. Only recompilation with the new settings will be needed in order to produce an updated program image compatible with the new platform. We were able to run the application successfully. The user button in this case was interfaced with one of the onboard pushbuttons.

Besides porting applications onto new platforms, reusability is also a key aspect. Sometimes, only a hardware modification is needed. For instance, a distance sensor can be used instead of a light sensor while preserving the same behavior. In this case, the robot would turn when an object is close to the distance sensor. In simulation, the distance sensor may be simulated and if the control unit stays the same, the models might be deployed as is without any change, only the sensor pDriver method would need to be adapted. The video in [119] shows the result obtained with the FRDM-64K and with a distance sensor.

6.8 Adding Connectivity: An IoT application

New features are constantly in demand for embedded products. Connectivity is particularly trending and at the core of Internet of Things (IoT). We have extended the line tracking robot to send a notification to a remote application whenever a roadblock is detected on the path. In order to achieve this, we added an obstacle sensor to detect the presence of an obstruction on the path, and added a WiFi shield in order to be able to send notifications to a remote application. The modifications made to the model hierarchy are shown in figure 6.4. The sensor unit has an additional sensor and the sensor controller has been wired to the cloud component in order to send notifications to the cloud.

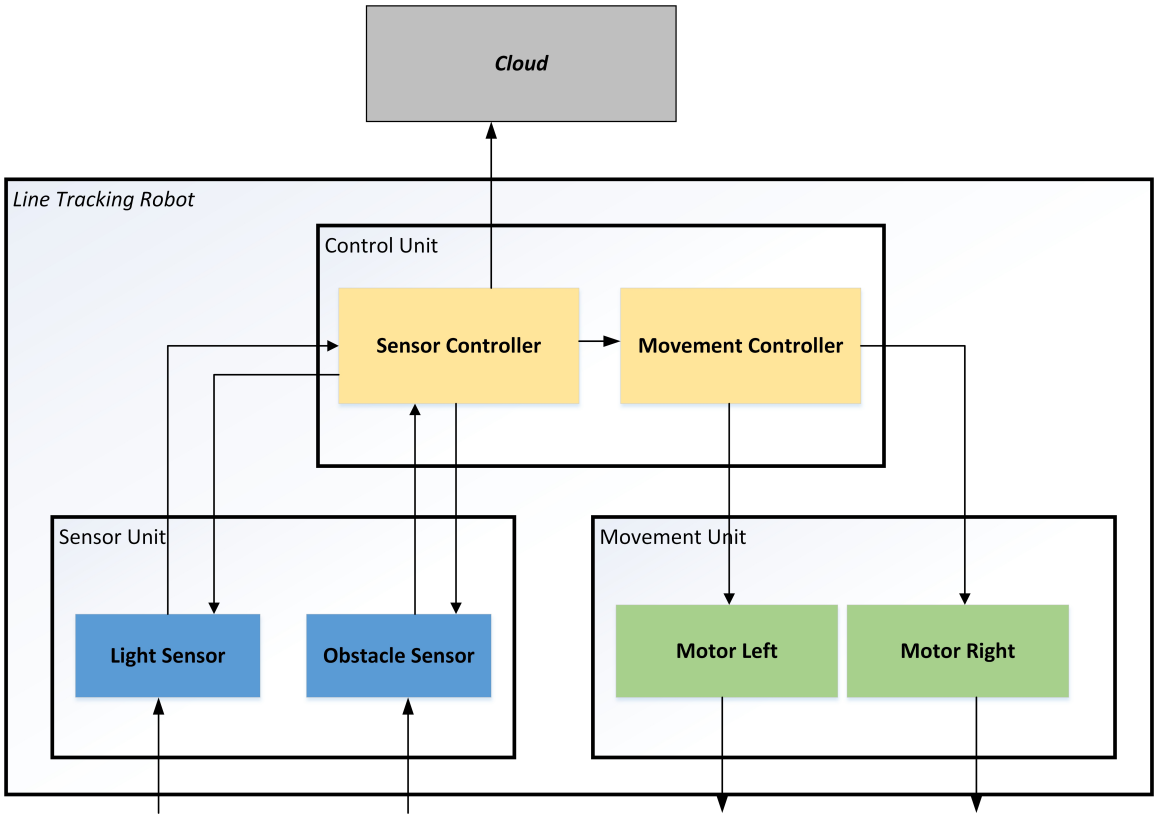

Figure 6.4: Modified Model Hierarchy Diagram

This application particularly uses the AT&T IoT cloud services [120], a platform for connecting devices to the Internet of Things. The data can also be stored using the M2X AT&T cloud-based data storage service and used to retrieve the time and location where a barrier was located by the robot. This could namely be used to alert emergency vehicles that a path is blocked. The sensor controller was modified to take into account the input of the obstacle of the sensor and to send output to the cloud application. More specifically, to enable the roadblock alert feature, upon the reception of an obstacle signal, the sensor controller external function transitions into a state that causes the output function to send a stop signal to the movement controller and a barrier notification to the cloud. This latter being connected to the sensor controller as an output port. In the *pDriver* method of the "cloud" port, we send the appropriate command to the AT&T IoT services to update the values and location. A video is available at [121] for demonstration purposes.

Chapter 7

Conclusion and Future Work

With the tremendous increase of the use of embedded systems and new application market demands, effective development practices ought to be applied in order to reduce the productivity gap. Model-based approaches were found to be the most promising solution to lessen the gap while improving the quality, correctness, and modularity of systems. We particularly used DEVS — an M&S formalism that has proven to be successful in real-time and embedded systems modeling — and introduced DEMES to illustrate DEVS-driven development of embedded systems. One important step of the development cycle is the transition from simulation platform to execution platform, i.e. from simulated models to executable model, in order to run models on the target hardware while preserving model continuity.

Using model-driven development for embedded systems is certainly a promising solution since the complexity and heterogeneity of the system are handled earlier in the development cycle. DEVS, in particular, with its formal nature and integrated time concept captures the essential characteristics of embedded systems. In this dissertation, we presented two DEVS-based real-time kernels that allow models to be executed on a variety of target platforms, and without the need of an operating system. Both kernels provide features similar to real-time kernels where formal models

act as system process, and event scheduling based on PDEVS algorithms. The first kernel allow previous CD++ and E-CD++ models to run on bare-metal and especially on ARM microcontrollers. The second kernel extends CDBoost with real-time execution and hardware-in-the-loop capabilities by using physical time in the PDEVS scheduling algorithm implementation, adding the port and driver concepts to interface with hardware components. The two kernels particularly differ in the communication mechanism used between model execution engines. These kernels are particularly essential since the model-driven development concept is built around transforming a model of a system into the real thing; and these kernels are what make that process possible in this context.

In addition to the new bare-metal executives, we presented a hardware abstract layer built around the MBED API, a vendor-independent library that allows us to easily run the model execution engines on multiple devices. This hardware abstract layer provides user-friendly commands to access hardware peripheral components and enable fast prototyping. There is also a wide community of MBED developers, and several libraries can be reused for interfacing models with various components. Another important aspect is the IoT opportunities that comes with the use of such API. Indeed, multiple IoT platform providers support MBED and through these services we are able to easily connect the developed models.

A case study was also presented to provide a practical view of the development cycle and the usability of the new bare-metal kernels. The line tracking robot was run on the E-CD++ kernel and using the Embedded CDBoost kernel. Embedded CDBoost particularly allowed us to have a small footprint and reduce the message processing overhead by more than 60%. We also showed how porting the application to a different platform and adding features is easy. In addition, we demonstrated

connectivity possibilities. This type of connectivity is also central to cyber physical systems, a rapidly growing field. The cyber physical technology is built on the embedded systems discipline and integrates computation, networking and physical processes.

Our work further extends the applicability of model-driven development by providing OS independent DEVS real-time executives that allow original models deployment onto multiple target hardware. The resulting DEVS firmware can be deployed onto multiple platforms. We have also introduced model connection to IoT platform and opened doors to future applications related to big simulation/data easily in order to fit the current trends. This latter aspect addresses one of the key issues that prevent M&S from achieving a wide-scale impact.

In the future we will address various aspects derived from this research. For instance, performing an exhaustive real-time analysis and measuring the impact of the peripheral library choice. This might be useful as — although not abstract and involving knowing the target hardware — snippet libraries might perform faster.

Another interesting path to explore is connecting IoT enabled devices to the M&S cloud services under development in our lab. This could allow to use simulation services as well.

References

[1] Q. Li and C. Yao, *Real-time concepts for embedded systems*. CRC Press, 2003.

[2] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *FM 2006: Formal Methods*, pp. 1–15, Springer, 2006.

[3] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Readings in hardware/software co-design*, vol. 86, 2001.

[4] M. Moallemi and G. Wainer, "Modeling and simulation-driven development of embedded real-time systems," *Simulation Modelling Practice and Theory*, vol. 38, pp. 115–131, 2013.

[5] G. Wainer and R. Castro, "DEMES: a Discrete-Event methodology for Modeling and simulation of Embedded Systems," *Modeling and Simulation Magazine, April*, vol. 2, pp. 67–73, 2011.

[6] D. Niyonkuru and G. Wainer, "Discrete Event Methodology for Embedded Systems," *Computing in Science & Engineering*, vol. 0, no. 0, pp. 2–13, 2015.

[7] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[8] Y. H. Yu and G. Wainer, "eCD++: an engine for executing DEVS models in embedded platforms," in *Proceedings of the 2007 summer computer simulation conference*, pp. 323–330, Society for Computer Simulation International, 2007.

[9] D. Vicino, D. Niyonkuru, G. Wainer, and O. Dalle, "Sequential PDEVS Architecture," in *Proceedings of the 2015 Spring Simulation Multiconference*, pp. 903–913, Society for Computer Simulation International, 2015.

[10] R. Toulson and T. Wilmshurst, *Fast and effective embedded systems design: applying the ARM mbed*. Elsevier, 2012.

[11] D. Niyonkuru and G. Wainer, "Towards a DEVS-based Operating System," in *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation*, pp. 101–112, ACM, 2015.

[12] A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, "Introduction: Modeling, Analysis and Synthesis of Embedded Software and Systems," in *Embedded Systems Development*, pp. 1–16, Springer, 2014.

[13] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[14] I. Radojevic and Z. Salcic, "Introduction," in *Embedded Systems Design Based on Formal Models of Computation*, pp. 1–6, Springer Netherlands, 2011.

[15] F. Herrera, H. Posadas, P. Sánchez, and E. Villar, "Systematic embedded software generation from systemc," in *Embedded Software for SoC*, pp. 83–93, Springer, 2003.

[16] S. Y. Liao, "Towards a new standard for system-level design," in *Hardware/-Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*, pp. 2–6, IEEE, 2000.

[17] G. Hu, S. Ren, and X. Wang, "A comparison of c/c++-based software/hardware co-design description languages," in *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pp. 1030–1034, Nov 2008.

[18] A. A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, no. 2, pp. 63–69, 2005.

[19] T. Henzinger, J. Sifakis, *et al.*, "The discipline of embedded systems design," *Computer*, vol. 40, no. 10, pp. 32–40, 2007.

[20] E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini, "SystemC/C-based model-driven design for embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 4, p. 30, 2009.

[21] G. Martin, "UML for embedded systems specification and design: motivation and overview," in *Proceedings of the conference on Design, automation and test in Europe*, p. 773, IEEE Computer Society, 2002.

[22] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," tech. rep., DTIC Document, 2006.

[23] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[24] H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, *Model Based Engineering of Embedded Real Time Systems*. Internat. Begegnungs-und Forschungszentrum für Informatik, 2007.

[25] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 0025–31, 2006.

[26] C. Atkinson and T. Kühne, "Model-driven development: a metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[27] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-driven architecture," in *Advances in Object-Oriented Information Systems*, pp. 290–297, Springer, 2002.

[28] O. M. Group, "MDA." http://www.omg.org/mda/. (Visited on 07/01/2015).

[29] P. Alexander and C. Kong, "Rosetta: Semantic support for model-centered systems-level design," *Computer*, vol. 34, no. 11, pp. 64–70, 2001.

[30] M. A. Khan, S. Saeed, A. Darwish, and A. Abraham, *Embedded and Real Time System Development: A Software Engineering Perspective: Concepts, Methods and Principles*, vol. 520. Springer, 2013.

[31] S. J. Mellor, T. Clark, and T. Futagami, "Model-driven development: guest editors' introduction.," *IEEE software*, vol. 20, no. 5, pp. 14–18, 2003.

[32] MathWorks, "MathWorks - MATLAB and Simulink for Technical Computing." http://www.mathworks.com/. (Visited on 07/01/2015).

[33] IBM, "IBM - IBM - Rational Rhapsody family." http://www-03.ibm.com/software/products/en/ratirhapfami. (Visited on 07/02/2015).

[34] E. Technologies, "SCADE Suite — Esterel Technologies." http://www.esterel-technologies.com/products/scade-suite/. (Visited on 07/01/2015).

[35] T. M. A. et al, "Modelica Tools Modelica Association." https://www.modelica.org/tools. (Visited on 07/01/2015).

[36] N. I. Corporation, "LabVIEW System Design Software - National Instruments." http://www.ni.com/labview/. (Visited on 07/01/2015).

[37] "Ptolemy Project Home Page." http://ptolemy.eecs.berkeley.edu/. (Visited on 07/01/2015).

[38] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu, "A next-generation design framework for platform-based design," in *Conference on using hardware design and verification languages (DVCon)*, vol. 152, 2007.

[39] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *Software, IEEE*, vol. 31, no. 3, pp. 79–85, 2014.

[40] MathWorks, "Simulink - Simulation and Model-Based Design." http://www.mathworks.com/products/simulink/index.html. (Visited on 07/01/2015).

[41] MathWorks, "DO-178C/DO-331 Checks - MATLAB & Simulink." http://www.mathworks.com/help/slvnv/ref/do-178c-checks.html?s_tid=srchtitle. (Visited on 07/01/2015).

[42] MathWorks, "Features - IEC Certification Kit." http://www.mathworks.com/products/iec-61508/features.html?refresh=true. (Visited on 07/01/2015).

[43] MathWorks, "Features - Simulink Verification and Validation." http://www.mathworks.com/products/simverification/features.html. (Visited on 07/01/2015).

[44] A. Tiwari, N. Shankar, and J. Rushby, "Invisible formal methods for embedded control systems," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 29–39, 2003.

[45] C. Eles and M. Lawford, *A tabular expression toolbox for matlab/simulink*. Springer, 2011.

[46] G. Hamon and J. Rushby, "An operational semantics for Stateflow," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 447–456, 2007.

[47] G. Simko, *Formal Semantic Specification of Domain-Specific Modeling Languages for Cyber-Physical Systems*. PhD thesis, Vanderbilt University, 2014.

[48] Y. Vanderperren and W. Dehaene, "From UML/SysML to Matlab/Simulink: current state and future perspectives," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pp. 93–93, European Design and Automation Association, 2006.

[49] O. M. Group, "OMG SysML." http://www.omgsysml.org/. (Visited on 07/01/2015).

[50] O. M. Group, "The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems — www.omgwiki.org/marte." http://www.omgmarte.org/. (Visited on 07/01/2015).

[51] O. M. Group, "MARTE 1.1." http://www.omg.org/spec/MARTE/1.1/. (Visited on 07/01/2015).

[52] A. Koudri, A. Cuccuru, S. Gerard, and F. Terrier, "Designing heterogeneous component based systems: evaluation of MARTE standard and enhancement proposal," in *Model Driven Engineering Languages and Systems*, pp. 243–257, Springer, 2011.

[53] S. Bernardi, J. Merseguer, and D. C. Petriu, "A dependability profile within MARTE," *Software & Systems Modeling*, vol. 10, no. 3, pp. 313–336, 2011.

[54] M. Ammar, M. Baklouti, and M. Abid, "Extending MARTE to support the specification and the generation of data intensive applications for massively parallel SoC," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pp. 715–722, IEEE, 2012.

[55] I. R. Quadri, A. Bagnato, and A. Sadovykh, "MADES EU FP7 Project: Model-Driven Methodology for Real Time Embedded Systems," in *Embedded and Real Time System Development: A Software Engineering Perspective*, pp. 57–89, Springer, 2014.

[56] H. Espinoza, D. Cancila, S. Gérard, and B. Selic, "Using MARTE and SysML for Modeling Real-Time Embedded Systems," *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, pp. 105–137, 2010.

[57] B. Selic and S. Gérard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Elsevier, 2013.

[58] H. Espinoza, D. Cancila, B. Selic, and S. Gérard, "Challenges in combining SysML and MARTE for model-based design of embedded systems," in *Model Driven Architecture-Foundations and Applications*, pp. 98–113, Springer, 2009.

[59] O. M. Group, "FUML." http://www.omg.org/spec/FUML/. (Visited on 07/01/2015).

[60] O. M. Group, "ALF." http://www.omg.org/spec/ALF/. (Visited on 07/01/2015).

[61] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier, "Formalizing Execution Semantics of UML Profiles with fUML Models," in *Model-Driven Engineering Languages and Systems*, pp. 133–148, Springer, 2014.

[62] F. Ciccozzi, A. Cicchetti, and M. Sjodin, "Towards translational execution of action language for foundational uml," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pp. 153–160, IEEE, 2013.

[63] R. George and P. Samuel, "Foundational UML Behavioral Specification with Java," *Procedia Computer Science*, vol. 46, pp. 941–948, 2015.

[64] A. F. Pires, T. Polacsek, V. Wiels, and S. Duprat, "Behavioural verification in embedded software, from model to source code," in *Model-Driven Engineering Languages and Systems*, pp. 320–335, Springer, 2013.

[65] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, 2003.

[66] P. C. Ölveczky, "Formal model engineering for embedded systems using real-time maude," *arXiv preprint arXiv:1107.0063*, 2011.

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

[68] M. Moallemi, G. Wainer, A. Awad, *et al.*, "Application of RT-DEVS in Military," in *Proceedings of the 2010 Spring Simulation Multiconference*, p. 29, Society for Computer Simulation International, 2010.

[69] A. Furfaro and L. Nigro, "A development methodology for embedded systems based on RT-DEVS," *Innovations in Systems and Software Engineering*, vol. 5, no. 2, pp. 117–127, 2009.

[70] X. Hu and B. P. Zeigler, "Model continuity to support software development for distributed robotic systems: A team formation example," *Journal of Intelligent and Robotic Systems*, vol. 39, no. 1, pp. 71–87, 2004.

[71] A. Furfaro and L. Nigro, "Embedded control systems design based on RT-DEVS and temporal analysis using UPPAAL," in *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pp. 601–608, IEEE, 2008.

[72] S. Mittal and J. L. R. Martín, *Netcentric system of systems engineering with DEVS unified process*. CRC Press, 2013.

[73] D. E. Thomas and P. R. Moorby, *The Verilog® Hardware Description Language*, vol. 2. Springer Science & Business Media, 2002.

[74] L. Capocchi, F. Bernardi, D. Federici, and P. Bisgambiglia, "Transformation of VHDL descriptions into DEVS models for fault modeling," in *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, vol. 2, pp. 1205–1210, IEEE, 2003.

[75] D. Huang and H. Sarjoughian, "Software and simulation modeling for real-time software-intensive systems," in *Distributed Simulation and Real-Time Applications, 2004. DS-RT 2004. Eighth IEEE International Symposium on*, pp. 196–203, IEEE, 2004.

[76] X. Hu and B. P. Zeigler, "Model continuity in the design of dynamic distributed real-time systems," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 35, no. 6, pp. 867–878, 2005.

[77] S. Hong and T. G. Kim, "Embedding UML subset into object-oriented DEVS modeling process," *SIMULATION SERIES*, vol. 36, no. 4, p. 161, 2004.

[78] M. Nikolaidou, V. Dalakas, and D. Anagnostopoulos, "Integrating Simulation Capabilities in SysML using DEVS," in *Proceedings of IEEE Systems Conference*, 2010.

[79] J. L. Risco-Martín, M. Jesús, S. Mittal, and B. P. Zeigler, "eUDEVS: Executable UML with DEVS theory of modeling and simulation," *Simulation*, vol. 85, no. 11-12, pp. 750–777, 2009.

[80] M. Nikolaidou, V. Dalakas, L. Mitsi, G.-D. Kapos, and D. Anagnostopoulos, "A SysML profile for classical DEVS simulators," in *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pp. 445–450, IEEE, 2008.

[81] G. A. Wainer, E. Glinsky, and P. MacSween, "A model-driven technique for development of embedded systems based on the DEVS formalism," in *Model-driven Software Development*, pp. 363–383, Springer, 2005.

[82] H. Saadawi and G. Wainer, "From DEVS to RTA-DEVS," in *Distributed Simulation and Real Time Applications (DS-RT), 2010 IEEE/ACM 14th International Symposium on*, pp. 207–210, IEEE, 2010.

[83] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*, pp. 200–236, Springer, 2004.

[84] "Uppaal." http://www.uppaal.org/. (Visited on 07/01/2015).

[85] H. Saadawi, G. Wainer, and M. Moallemi, "Principles of DEVS models verification for real-time embedded applications," 2011.

[86] Y. K. Cho, X. Hu, and B. P. Zeigler, "The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems," *Simulation*, vol. 79, no. 4, pp. 197–210, 2003.

[87] F. Bergero and E. Kofman, "PowerDEVS: a tool for hybrid system modeling and real-time simulation," *Simulation*, vol. 87, no. 1-2, pp. 113–132, 2011.

[88] H. S. Song and T. G. Kim, "Application of real-time DEVS to analysis of safety-critical embedded control systems: railroad crossing control example," *Simulation*, vol. 81, no. 2, pp. 119–136, 2005.

[89] M. Moallemi, G. Wainer, F. Bergero, and R. Castro, "Component-oriented interoperation of real-time DEVS engines," in *Proceedings of the 44th Annual Simulation Symposium*, pp. 127–134, Society for Computer Simulation International, 2011.

[90] M. Moallemi and G. Wainer, "A system-on-chip FPGA implementation of embedded CD++," in *Proceedings of the 2009 Spring Simulation Multiconference*, p. 153, Society for Computer Simulation International, 2009.

[91] R. Castro, I. Ramello, M. Bonaventura, and G. A. Wainer, "M&S-based design of embedded controllers on network processors," in *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, p. 32, Society for Computer Simulation International, 2012.

[92] S. M. Cho and T. G. Kim, "Real-Time DEVS simulation: concurrent time-selective execution of combined RT-DEVS and interactive environment," *SCSC-98*, pp. 410–415, 1998.

[93] X. Hu, B. Zeigler, and J. Couretas, "Devs-On-A-Chip: implementing DEVS in embedded java on a tiny internet interface for scalable factory automation," in *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*, pp. 3051–3056, 2001.

[94] "Xenomai — real-time framework for linux." http://xenomai.org/. (Visited on 07/01/2015).

[95] G. Wainer, "Cd++: a toolkit to develop devs models," *Software: Practice and Experience*, vol. 32, no. 13, pp. 1261–1306, 2002.

[96] G. Wainer and E. Glinsky, "Model-based development of embedded systems with RT-CD++," in *Proceedings of the WIP Session, IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, ON, Canada*, 2004.

[97] M. Moallemi and G. Wainer, "Designing an interface for real-time and embedded DEVS," in *Proceedings of the 2010 Spring Simulation Multiconference*, p. 137, Society for Computer Simulation International, 2010.

[98] A. C. Chow, "Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator," *TRANSACTIONS of the Society for Computer Simulation*, vol. 13, no. 2, pp. 55–68, 1996.

[99] J. Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2013.

[100] ARM, "Processors - ARM." http://www.arm.com/products/processors. (Visited on 07/21/2015).

[101] ARM, "Cortex-M3 Processor - ARM." http://www.arm.com/products/processors/cortex-m/cortex-m3.php. (Visited on 07/21/2015).

[102] ARM, "Cortex-M4 Processor - ARM." http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php. (Visited on 07/21/2015).

[103] "CMSIS - Cortex Microcontroller Software Interface Standard - ARM." http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php. (Visited on 07/21/2015).

[104] STMicroelectronics, "STM32Snippets - STMicroelectronics." http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1898?sc=stm32snippets. (Visited on 07/21/2015).

[105] STMicroelectronics, "STM32 Embedded Software." http://www.st.com/st-web-ui/static/active/en/resource/sales_and_marketing/presentation/product_presentation/stm32_embedded_software_offering.pdf. (Visited on 08/10/2015).

[106] STMicroelectronics, "STM32CubeMX." http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1898?sc=stm32snippets. (Visited on 07/21/2015).

[107] A. MBED, "mbed library internals - Handbook — mbed." https://developer.mbed.org/handbook/mbed-library-internals. (Visited on 07/21/2015).

[108] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.

[109] E. Glinsky and G. Wainer, "DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments," in *Distributed Simulation and Real-Time Applications, 2005. DS-RT 2005 Proceedings. Ninth IEEE International Symposium on*, pp. 265–272, Oct 2005.

[110] "Boost C++ Libraries." http://www.boost.org/. (Visited on 07/01/2015).

[111] A. S. Eddin, "Modelling and simulation of a line tracking robot with rt-devs," tech. rep., Carleton University, Ottawa, Jan 2013.

[112] Pololu, "Pololu - QTR-1RC Reflectance Sensor." https://www.pololu.com/product/959. (Visited on 07/01/2015).

[113] A. R. S. Laboratory, "Line Tracking Robot on Lego Hardware 1 - YouTube." https://www.youtube.com/watch?v=mTtlSV7WbuI, January 2013. (Visited on 03/01/2015).

[114] A. R. S. Laboratory, "Line Tracking Robot on the Disco Board(Early Debug Version)- YouTube." https://www.youtube.com/watch?v=X2itlznkoVw, February 2015. (Visited on 07/21/2015).

[115] A. R. S. Laboratory, "Line Tracking Robot on a STM32 Nucleo Board - YouTube." https://www.youtube.com/watch?v=pco-wjaCPIA&feature=youtu.be, July 2015. (Visited on 07/21/2015).

[116] A. R. S. Laboratory, "Line Tracking Robot on Lego Hardware 2- YouTube." https://www.youtube.com/watch?v=None_qX4go0&feature=youtu.be, January 2013. (Visited on 03/01/2015).

[117] A. R. S. Laboratory, "Line Tracking Robot - No Line detected - YouTube." https://www.youtube.com/watch?v=sDvFz1cgYDo&feature=youtu.be, July 2015. (Visited on 08/11/2015).

[118] A. R. S. Laboratory, "Line Tracking Robot - Embedded CDBoost - YouTube." https://www.youtube.com/watch?v=BZzzeJAa-cA&feature=youtu.be, August 2015. (Visited on 08/01/2015).

[119] A. R. S. Laboratory, "Obstacle Tracking on The Freedom Board - YouTube." https://www.youtube.com/watch?v=1Pnsi3sezXU&feature=youtu.be, August 2015. (Visited on 08/03/2015).

[120] AT&T, "AT&T M2X: Data storage for the Internet of Things." https://m2x.att.com/. (Visited on 08/11/2015).

[121] A. R. S. Laboratory, "IoT Demo - Extending the Line Tracking Robot - YouTube." https://www.youtube.com/watch?v=jNOAOCMQ5hc&feature=em-share_video_user, August 2015. (Visited on 08/11/2015).

[122] KEIL, "Keil MCBSTM32F200 Evaluation Board Overview." http://www.keil.com/mcbstm32f200/. (Visited on 07/21/2015).

[123] STMicroelectronics, "32F429IDISCOVERY Discovery kit with STM32F429ZI MCU - STMicroelectronics." http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF259090. (Visited on 07/21/2015).

[124] STMicroelectronics, "NUCLEO-F411RE STM32 Nucleo development board with STM32F411RET6 MCU, supports Arduino - STMicroelectronics." http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/LN1847/PF260320. (Visited on 07/21/2015).

[125] KEIL, "Keil Embedded Development Tools for ARM, Cortex-M, Cortex-R4, 8051, C166, and 251 processor families.." http://www.keil.com/. (Visited on 07/21/2015).

[126] STMicroelectronics, "STM32F207IG High-performance ARM Cortex-M3 MCU with 1 Mbyte Flash, 120 MHz CPU, ART Accelerator, Ethernet - STMicroelectronics." http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1575/LN9/PF245085?sc=internet/mcu/product/245085.jsp. (Visited on 07/21/2015).

[127] STMicroelectronics, "STM32F429ZI High-performance advanced line, ARM Cortex-M4 core with DSP and FPU, 2 Mbytes Flash, 180 MHz CPU, ART Accelerator, Chrom-ARTAccelerator, FMC with SDRAM, TFT - STMicroelectronics." http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1577/LN1806/PF255419. (Visited on 07/21/2015).

[128] STMicroelectronics, "STM32F411RE High-performance access line, ARM Cortex-M4 core with DSP and FPU, 512 Kbytes Flash, 100 MHz CPU, ART Accelerator - STMicroelectronics." http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1577/LN1877/PF260049. (Visited on 07/21/2015).

[129] A. MBED, "mbed — welcome." http://mbed.org/. (Visited on 07/21/2015).

Appendix A

The DEVS Formalism

The DEVS formalism decomposes complex system designs into basic (behavioral) models called *atomic* and composite (structural) models called *coupled* [67]. A coupled model is composed of a group of atomic and/or coupled models with well-defined coupling connections between its components.

Atomic Model

In the atomic model, an incoming input event x triggers an external transition δ_{ext}. This transition is a function of the current state s, the input event x (set of ports and values) and elapsed time e since the last transition of the system. Based on the conditions of x and e, the external transition changes the state of the system to $s\prime$. Inside the system, there could be a time advance timer set to trigger when its value ta has elapsed since it was reset in the last transition. This trigger simultaneously causes an internal transition δ_{int} and output function λ, both functions of the current state s. The output function generates an output event y and the internal transition changes the current system state to a new state $s\prime$ as shown in figure A.1.

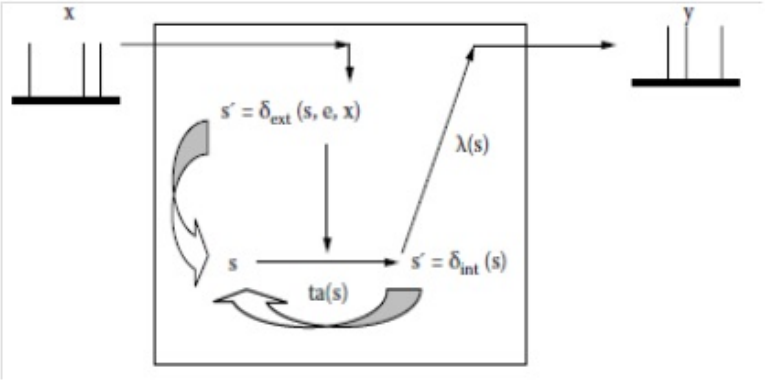

Figure A.1: Informal Definition of an Atomic Model.

Coupled Model

The DEVS specification of a coupled model is different from that of an atomic model as it doesnt define system state transitions or output functions but instead defines the formal port connections and their directions between the components. The following figure is a simple example of a coupled model hierarchy:

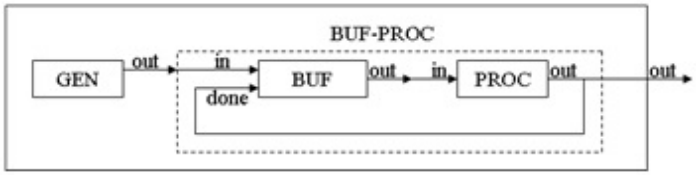

Figure A.2: Generator-Buffer-Processor Hierarchical DEVS Model.

The Top model is a coupled model made of an atomic model GEN and a coupled model BUF-PROC, connected via the GEN::out → BUF-PROC::in connection. BUF-PROC is connected to the Top model via the BUF-PROC::out → Top::out connection. BUF-PROC is also composed of two atomic models BUF and PROC whose ports are interconnected as shown in the figure above.

A.1 Formal Specification

A DEVS atomic model is formally defined by:

$$AM = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle,$$

Where:

X = (p,v) | p ∈ IPorts, v ∈ X_p is the set of input ports and values;

Y = (p,v) | p ∈ OPorts, v ∈ Y_p is the set of output ports and values;

S : is the set of sequential states;

δ_{int}: S → S is the internal state transition function;

δ_{ext}: Q × X → S is the external state transition function, where:

Q = {(s,e) | s ∈ S, $0 < \in <$ ta(s)} is the total state set, e is the time elapsed since the last state transition;

λ: S → Y is the output function;

ta: S → $R^+_{0,\infty}$ is the time advance function.

A DEVS coupled model is formally defined by:

$$CM = \langle X, Y, D, \{Md | d \in D\}, EIC, EOC, IC, select \rangle$$

Where:

X = {(p,v) | p ∈ IPorts, v ∈ X_p} is the set of input events, where IPorts represents the set of input ports and X_p represents the set of values for the input ports;

Y = {(p,v) | p ∈ OPorts, v ∈ Y_p} is the set of output events, where OPorts represents the set of output ports and Y_p represents the set of values for the output ports;

D is the set of the component names and for each d ∈ D;

Md is a DEVS basic (i.e., atomic or coupled) model;

EIC is the set of external input couplings, EIC $\subseteq$ {((Self, inSelf), (j, inj)) | inSelf $\in$ IPorts, j $\in$ D, inj $\in$ Iportsj};

EOC is the set of external output couplings, EOC $\subseteq$ {((i,outi),(Self,outSelf)) | outSelf $\in$ OPorts, i $\in$ D,outi $\in$ OPortsi};

IC is the set of internal couplings, IC $\subseteq$ {((i,outi), (j,inj)) | i,j $\in$ D, outi $\in$ OPortsi , inj $\in$ IPortsj};

select is the tiebreaker function, where select $\subseteq$ D $\rightarrow$ D, such that, for any nonempty subset E, select (E) $\in$ E.

Appendix B

Development boards and Software Development Flow

Three types of boards were used to develop DEVS bare-metal applications: The MCSTM32F200 Evaluation Board [122], the STM32F429 Discovery Board [123] and the Nucleo-F411RE Board [124]. We will respectively refer to them as the Eval board, the Disco board and the Nucleo board for reasons of brevity.

B.1 Development Boards

B.1.1 KEIL's MCSTM32F200 Evaluation Board

KEIL [125] designs and manufactures evaluation boards to help evaluate a new MCU architecture. The MCSTM32F200, show in figure 3, has a 120MHz STM32F207IG [126] ARM Cortex-M3 processor-based MCU. This board also comes with on-chip memory (1MB Flash and 128KB RAM) and external memory (namely 2MB SRAM, 8MB NOR Flash and 8KB I2C EEPROM).

Other features include a 2.4-inch color LCD touchscreen, 10//100 Ethernet Port,

two USB 2.0 (Full and high speed) ports, a serial/UART port, a MicroSD Card Interface, a 5-position joystick, a 3-axis digital accelerometer, a 3-axis digital gyroscope, a potentiometer, an audio CODEC with line-in/out and speaker/microphone, a digital microphone, a digital VGA camera, pushbuttons, LEDs, power supply jacks, and debug interface connectors.

B.1.2 STM32F429 Discovery Board

The disco board has a 180 MHz STM32F429ZIT6 MCU (with a Cortex-M4 processor, 2MB of Flash, 256KB of RAM) [127], includes an embedded debug tool, a 2.4-inch LCD, 64MB external SDRAM, a gyroscope, a USB micro connector, LEDs and pushbuttons.

B.1.3 NUCLEO-F411RE Board

This Nucleo board has a STM32F411RET6 MCU (with a 100 MHz ARM Cortex-M4 CPU, 512KB of flash and 128KB of RAM) [128]. This board particularly offers Arduino connectivity (makes it possible to use existing Arduino shields) and ST Morpho connectors. It is also MBED enabled [129], i.e. implements the MBED HDK, has an on-board debugger/programmer, LEDs, pushbuttons, and USB capabilities with three different interfaces (Virtual COM port, Mass Storage and Debug Port).

Shields used with the Nucleo Board

We have used some Arduino compatible shields and connected them to/plugged the Nucleo: the Parallax Robotic Shield and the Seeed Studio Shield. These shields comes with onboard sensors (such as infrared sensors for the Seeed shield bot and touch sensors for the Parallax robotic shield), motors and offer a mobile platform

that makes robotic development easy. The Nucleo board acts as the brain and sends different commands to the shields components in order to get sensor values or send actuators commands.

B.2 Embedded Software Development

Developing software for the previous boards requires several steps. Figure B.1 [99] shows a simplified development flow. A new project is created in an IDE (Eclipse in our case), and then driver libraries from the MCU vendor, application code (DEVS models and real-time executive in our case) and optional middleware (e.g. RTOS) files are added to the project. Diverse project options can be setup (e.g. debug mode activation, compiler optimization,...). Then, all the source code is compiled, linked, and then the program image is uploaded to the flash memory if the build is successful.

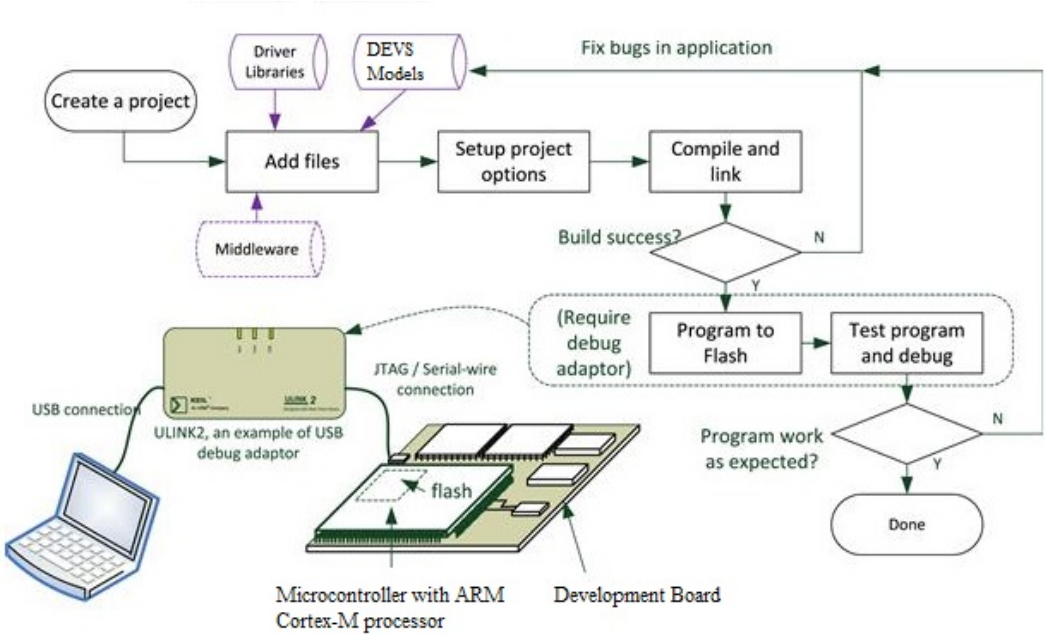

Figure B.1: Simplified Software Development Flow

The program behavior is then observed to see if it corresponds to the expected response. If debug is required, a debug adapter can be used and debugging operations (e.g. software tracing, stepping through code . . .) performed. When no real hardware is available, the executable image can be tested by simulation using an instruction set simulator. We have integrated QEMU to our Eclipse IDE to provide this capability.

Appendix C

Formal Specification of the Case Study

We will show in this appendix the control unit coupled model and the sensor controller atomic model formal specification.

C.1 Example of a CM formal specification - The Control Unit

As mentioned earlier, the control unit has two atomic models, the sensor and movement controllers. The control unit can be formally defined as (see section 2.3 for details):

$$CM = \langle X, Y, D, \{Md\}, EIC, EOC, IC \rangle,$$

Where

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

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

D={Sensor Controller,Movement Controller}

Md={M(Sensor Controller), M(Movement Controller)}

EIC={((Self,CU_START_IN_TOP),(Sensor Controller,sctrl_start_in));
((Self,CU_LIGHT_IN_SU),(Sensor Controller,sctrl_light_in))}
EOC={((Sensor Controller,sctrl_start_out),(Self,CU_START_OUT_SU));
((Movement Controller,mctrl_movel_out),(Self,CU_MOVEL_OUT_MU));
((Movement Controller, mctrl_mover_out),(Self,CU_MOVER_OUT_MU))}
IC={(Sensor Controller,sctrl_mctrl_out);(Movement Controller,mctrl_sctrl_in)}
In the above specification, X represents the set of input events (N being the set of port values); Y the set of output events; D the component name of each model; Md the DEVS basic (atomic or couple) model; and finally EIC(external input coupling), EOC (external output coupling), and IC (internal couplings) describe the port links between models.

C.2 Example of an AM formal specification - The Sensor Controller

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

$$M = \langle X, S, Y, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta \rangle,$$

Where
X:{(sctrl_light_in,{BRIGHT,DARK,ALL_DARK});
(sctrl_start_in,{START_PROC,STOP_PROC});(sctrl_mctrl_in, {})}
S:{“IDLE”, “PREP_RX”, “WAIT_DATA”, “TX_DATA”, “PREP_STOP”}
Y:{(sctrl_mctrl_out,{ON_TRACK, OFF_TRACK,STOP_PROC});
(sctrl_start_out,{START_PROC, STOP_PROC})}

```
δint(s) {

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

}
δconf(s,e,x){

    δint(s);
    e = 0;
    δext(s,e,x);

}
λ(s) {

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

```
                    else if(sensor_input == BRIGHT)
                                                output_val = OFF_TRACK;
                    sendOutput( time, sctrl_mctrl_out , output_val) ;
        }

}
```

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