Real-Time and Embedded Systems Development based on Discrete Event Modeling and Simulation

By

Mohammad Moallemi, B. Eng., M. A. Sc.

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE)
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

September 2011

The undersigned recommend to
the Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

Real-Time and Embedded Systems Development based on Discrete Event Modeling and Simulation

submitted by
Mohammad Moallemi, B. Eng., M. A. Sc.
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Electrical and Computer Engineering

_____

Chair, Howard Schwartz, Department of Systems and Computer Engineering

_____

Thesis Supervisor, Gabriel Wainer

_____

External Examiner, Andrea D'Ambrogio, Dept. of Business Engineering, University of Roma

Carleton University
September 2011

To my parents,
making me who I am now with their love and support,
and to my loving wife, Shafagh,
for his constant love, support, and encouragement.

Abstract

The design and development of embedded hard real-time (RT) systems is one of the most complex software development practices, because of the criticality and timeliness required for these systems. One critical aspect of RT systems is the production of outputs before the specified deadline. A late output caused by an overrun condition in the processing of RT tasks, not only degrades the system performance but also produces catastrophic results. Formal methods are promising alternatives in dealing with the design issues of these applications, however they do not scale well for complex systems. A cost-effective approach to verify the design and implementation details of such applications is the use of Modeling and Simulation (M&S). These methods provide dynamic and risk-free testing environments to verify different variable scenarios. M&S is now limited to feasibility analysis and verification of such systems, hence the simulation models are not used in the development of the final embedded application.

This dissertation is proposing an M&S-based method referred to as DEVSRT (Discrete EVent System Specifications in Real-Time) to solve the discontinuity between the simulation models and the final embedded software. The proposed approach combines the advantages of a simulation-based method and a formal methodology to develop embedded applications, and integrate simulation models with hardware components.

The research also proposes an integration of DEVSRT with Imprecise Computations theory. The proposed I-DEVS (imprecise DEVS) formalism uses a dynamic scheduling algorithm based on the criticality of the RT tasks to manage overload situations in the system by degrading the system's output accuracy in order to meet hard deadlines. The algorithm detects transient overloading conditions early enough to carry out a proper imprecise scheduling of RT tasks, providing a more reliable runtime platform.

Acknowledgements

I want to express my sincere gratitude toward my adviser and my mentor, Professor Gabriel Wainer, for his support, guidance, and trust that helped me through my graduate studies. His diligence and commitment to science have been and will be a great influence on me for many years to come. I am grateful for having the opportunity to learn from him and work with him.

I would also like to thank the members of the ARS Laboratory and the Department of Systems and Computer Engineering at Carleton University.

Table of Contents

Chapter 1: Introduction **1**

1.1 Contributions 6
1.2 Research Publications 8
1.3 Organization 11

Chapter 2: Background **13**

2.1 Modeling and Simulation Concepts in DEVS 14
2.2 Classical DEVS Formalism 17
2.3 Parallel DEVS Formalism 20
A) Abstract Simulation Algorithm 22
2.4 Real-Time DEVS Formalism 25
2.5 DEVS Simulation in CD++ 26
A) CD++ Software Architecture 26
B) DEVS Model Definition in CD++ 28
2.6 Modeling and Simulation-based Approaches 29
2.7 DEVS-Based Approaches 31
A) Model Continuity 33
B) Real-Time Deadline 35
C) Simultaneous Events 35

Chapter 3: The DEVSRT Formalism **37**

3.1 Real-Time Interface 41
3.2 Implementation on E-CD++ 44
A) E-CD++ Software Structure 47
B) Performance Evaluation 50

3.3 Case Study: e-puck Robot Controller 54

Chapter 4: Extended Applications of DEVSRT **65**

4.1 DEVS-Based Collaborative Modeling 65

A) Message Structure 67

B) Example Collaborative Model 68

4.2 DEVSRT and Visualization 73

A) Message Structure and Implementation 78

Chapter 5: Imprecise DEVS **81**

5.1 Algorithms for Imprecise Computation 83

5.2 DEVS Task Model 84

A) Problem Statement 86

5.3 I-DEVS Formalism 87

A) Example 89

5.4 Results and Discussions 92

A) Performance Evaluation 100

B) Scalability 101

Chapter 6: Conclusions and Future Work **102**

6.1 Review of the Contributions 103

6.2 Future Work 106

References **108**

List of Tables

Table 2.1: Comparing RT and Embedded DEVS Modeling Approaches. 34
Table 3.1: DEVS Output Mapping Table. 56

List of Figures

Figure 2.1: Entities in a DEVS-based M&S framework [Zei00] 14
Figure 2.2: M&S layers in a DEVS-based system (modified from [Wai09]) 16
Figure 2.3: DEVS atomic component state transition sequence (modified from [Wai09]) 18
Figure 2.4: Coupled DEVS model example [Wai09]. 20
Figure 3.1: DEVSRT Development Cycle (modified from [Wai11]). 39
Figure 3.2: E-CD++ with DEVSRT Development Framework. 46
Figure 3.3: E-CD++ software structure. 48
Figure 3.4: Synthetic Model Architecture (Modified from [Gli02]). 51
Figure 3.5: Percentage of Overhead with Variable Depth 52
Figure 3.6: Percentage of Overhead with Variable Width. 53
Figure 3.7: a) e-puck robot b) placement of sensors and LEDs. 54
Figure 3.8: epuck0 atomic component state diagram. 57
Figure 3.9: Atomic Animation diagram for e-puck random distance test. 61
Figure 3.10: event-file scenarios a) scenario 1 b) scenario 2. 62
Figure 3.11: Atomic Animation diagram for e-puck random distance test. 63
Figure 4.1: Overview of the Partitioned E-puck Model. 69
Figure 4.2: E-Puck Controller Collaborative DEVS Model. 70
Figure 4.3: E-CD++ input and output log files 72
Figure 4.4: Collaborative System Architecture [Moa11b]. 74
Figure 4.5: Detailed System Overview [Moa11b] 75
Figure 4.6: DEVS Graph of the robot controller [Moa11b].. 77
Figure 4.7: 3D Visualization Engine Zoomed Map [Moa11b]. 80
Figure 5.1: A Monotone Task Divided to Mandatory and Optional Parts [Liu94a]. 82
Figure 5.2: Processing Carried for a State Transition. 85
Figure 5.3: Overload Scenario. 86

Figure 5.4: Example I-DEVS Model. 90
Figure 5.5: Example Transient Overload Scenario. 92
Figure 5.6: Applying Imprecise Computation to the Sample Scenario. 92
Figure 5.7: Synthetic Robotic Model Used for Verification. 94
Figure 5.8: Discarded Tasks vs. Processor Utilization. 94
Figure 5.9: Response Time vs. Execution Time in Heavy Load. 95
Figure 5.10: Number of Discarded Tasks vs. Average Response Time in Medium Load. 96
Figure 5.11: Number of Discarded Tasks vs. Processor Utilization in Heavy Load. 97
Figure 5.12: Number of Components per Level vs. Average Response Time. 98
Figure 5.13: Number of Components per Level vs. Overhead Percentage. 99

Acronyms

DEVS	Discrete EVent System Specifications
P-DEVS	Parallel DEVS
RT-DEVS	Real-Time DEVS
M&S	Modeling and Simulation
RC	Root Coordinator
DEVSRT	Discrete EVent System Specifications in Real-Time
E-CD++	Embedded CD++
IDE	Integrated Development Environment
GGAD	Generic Graphical Advanced environment for DEVS modeling and simulation
IC	Imprecise Computations
I-DEVS	Imprecise DEVS
HSC	Hardware-Software Co-design
HIL	Hardware-In-the-Loop
HILS	Hardware-In-the-Loop Simulation
RT	Real-Time
SDE	Simulation-Driven Engineering

Chapter 1: Introduction

Real-time (RT) and embedded systems are employed in various applications ranging from telecommunications, customer electronics, transportation, medical equipments, and intelligent and automated systems. An RT system is defined by Liu [Liu00] as “a system that is required to complete its work and deliver its services on a timely basis”. In this definition, those systems in which all timing constrains must be met are considered “hard real-time systems”. Many of these systems are deployed in embedded microprocessors working in hardware computing platforms with special configurations and interfaces. [Nic10] describes an embedded system as “a system designed to perform a dedicated function, typically with tight real-time constraints, limited dimensions, and low cost and low-power requirements”. The architecture of these systems usually integrates different types of hardware components such as processors, analog and digital components, as well as mechanical (e.g. sensors and actuators) and visual components, which demands increasingly challenging multidisciplinary design and development efforts [Nic10]. Nevertheless, because of heterogeneity of these systems and their constraints (such as cost, time to market, and performance), their development cycle is time consuming, error prone and expensive.

In embedded systems with hard real-time constraints, the design decisions can lead to catastrophic consequences for infrastructures or lives [Liu00]. These days, many critical and complex real-time control systems rely completely on computer-based systems. Examples of these applications are flight control systems, automotive applications, telecommunication systems, railway switching, nuclear power plant control systems, traffic control systems, complex manufacturing systems, space missions, etc. The size, variety, and criticality of the computations carried out in these systems have attracted more abstract and visual design methods that increase their complexity, reliability and performance. On the other hand, heuristic, and ad-hoc design approaches lack flexibility, reusability, reliability and scalability in the final

system. They are also prone to tedious programming, difficult code understanding, software maintenance, and verification of time constraints issues [But10].

A solution proven to provide a reliable framework for designing these systems is the adoption of formal methods. Formal methods are special cases of mathematical-based techniques for design, development and verification of software and hardware systems [Mon03]. They allow for appropriate mathematical specification and analysis of the designs, which can contribute to the reliability of the final system, yet they add to the complexity of the design and increase the cost of the development. Hence, they are mostly appropriate for systems with critical applications where safety and robustness is a foremost aspect. Unfortunately, these methods do not scale up well, as most formal proving mechanisms cannot provide formal proofs of correctness when the complexity of the system grows [Fin96, and Abr06].

Instead, Modeling and Simulation (M&S) provides a practical solution in solving the above mentioned difficulties in the design of RT and embedded systems, caused by formal methods. Computer-based M&S is a useful tool for efficient analysis, design, verification and optimization of general dynamic systems. The use of M&S in software engineering reduces costs and risks and allows for exploring different aspects of the system.

Formal M&S is a branch of M&S, in which the simulation models are defined using a formal approach. This technique has shown promising results in making multidisciplinary system development tasks manageable [Zei00]. It provides a hierarchical design scheme in which higher abstract levels are branched into levels that contain more details. The system specifications are expressed using mathematical notations in which the details of the behavior of the system are accurately modeled. Other advantages of formal M&S are applying formal model checking techniques at design time [Son05, and Saa09], incremental refinement of the initial simulation models, simulation-based validation, reuse of the existing models, risk free testing of critical real-time applications.

The use of M&S is now popular in the early stages of RT and embedded systems development, because this method raises the abstraction level and gives a clear view of the

behavior of the system to be developed. However, when the scope of the development moves towards the actual target hardware, the early simulation models are abandoned and the final system is redeveloped from scratch, based on the results obtained during the simulation phase. Consequently, M&S is often used only for testing, verification, and feasibility analysis of these systems. Currently, existing development tools and methods do not support a simulation-based approach or they lack the model continuity concept [Hu04, Hu05]. Model continuity applied from the early simulation stages to the final target deployment, shortens the development process and speeds up the implementation phase. In this approach, M&S is not only a foremost component in the development, but also goes further by utilizing the simulated model as the final target architecture. Commercial tools such as MATLAB/Simulink [Cha09], and LabVIEW [Tra06] are mostly limited to simulation and do not directly support model continuity. In addition, approaches like UML-RT [Kus01] can be used to develop software design models, which are not suitable to be used as simulation models [Hua04].

Therefore, this research investigates the use of M&S-driven engineering at every step in the entire embedded RT application development phases (which includes design, development, testing, and deployment). M&S-driven engineering is a computational approach derived from M&S, exploring the use of simulation models in software development. The use of this method for this kind of application allows for testing in a simulated environment using virtual and real-time simulation with different test scenarios [Yu07a, Sha07], and incrementally deploying them in the target hardware.

The followings are the motivations in using formal M&S for design and development of RT and embedded systems:

- **Reliability and robustness**: a formal methodology provides a reliable mathematical framework for RT applications, in which structural representation of components and formal means for explicitly specifying timings are presented.
- **Hardware-Software Co-design (HSC)**: The drawbacks of independent design and development of hardware and software parts of an embedded system are in the complexity

of the integration of these two parts, in which ambiguous and unrecognized system malfunctions can be caused by incompatibilities between the software and hardware, producing longer and more expensive development cycle [Bal97]. Instead, a hierarchical and component-oriented M&S framework allows the designer to define the structure and behavior of the system using state machines and then replace them with hardware and software surrogates after performing simulation-based verification of the components. This method provides an integrated design platform for co-designing the hardware and software components together and later, deciding which component goes as hardware and which one as software.

- **Model reuse**: The component level encapsulation of behavior and data and well-defined coupling of components in these methodologies allow model designers to reuse the existing models. Model reuse leads to a faster model development, since many already available sub-models can be integrated with the new ones.
- **Knowledge reuse**: Many existing techniques that are popular in M&S of real-time and embedded systems - such as State Charts [Sch00], Verilog [Kim01], VHDL [Cap03], Petri Nets and Timed Petri Nets [Jac02], Timed Automata [Gia03], Finite State Machines [Zhe03], and DEVS [Zie00] – can be formally transformed into another method. This permits sharing model-level data, allowing for designing hybrid models with sub-models defined by different methodologies and a potential for collaborative and heterogeneous modeling.
- **Collaborative model execution**: Different behavioral components can cooperate at runtime via lightweight interfaces in which real-time models implemented in different tools communicate with each other. This collaborative approach, where the simulators themselves see each other as real-world devices permits the run-time model to collaborate with different models implemented on various simulators. It also allows for model encapsulation and employing the benefits offered by other tools (e.g. continuous system modeling [Cel06], virtual reality environments).

- **Hardware-In-the-Loop Simulation (HILS)**: One of the cost saving, efficient, and risk-free ways to develop an embedded system is to deploy HIL technique [Gli04a]. Integrated design of hardware controller with the plant under study as separate but interacting components allows for HIL test of the controller, while the plant components can be incrementally replaced with the actual segments, saving time and cost in the development of the entire system, as well as exploring dangerous and impractical situations.

The Discrete-EVent System Specification (DEVS) formalism [Zei00] has been chosen for this purpose for the following reasons. DEVS provides a formal foundation to M&S that proved to be successful in different complex applications [Wai09]. It integrates a simulation approach with the benefits of a formal modeling technique, which provides fast prototyping and incremental development while allowing for reuse of previously existing models. Concurrent with these theoretical advances, various DEVS-based simulation tools have been implemented, such as DEVS-C++ [Zei96], RTDEVS/CORBA [Cho03], DEVSCluster [Kim04], and DEVS/SOA [Mit09]. In particular, the CD++ toolkit [Wai02b] is an open-source, object-oriented M&S environment that implements DEVS formalism using different middleware technologies on various platforms [Wai04, Chi07, Liu07, Yu07a, Har08, Wai08a, Wai08b, and Wai09].

Aside from the above-mentioned motives, this dissertation intends to investigate techniques to overcome overrun conditions in hard-real-time systems designed with a DEVS-based M&S-driven approach. One critical aspect of a hard RT system is the production of outputs before the specified deadline. A late output in such systems not only degrades the system performance but also produces catastrophic results (loss of lives and expensive equipments). However, in circumstances with system overloads, it might be impossible to meet the deadlines. Since RT and embedded systems are not deterministic, tasks may enter the system at any time hence, there is no prior knowledge of their occurrence times [Liu00].

The Imprecise Computation (IC) technique [Liu94a] helps to overcome these high computation peaks by discarding unnecessary computations in overload conditions. The main idea is to separate the computation into mandatory and optional parts (the mandatory part affects

the correctness of the result and the optional affects its quality). This research aims at introducing a flexible RT task execution paradigm for DEVS by incorporating IC technique with the DEVS task model. Based on the requirements in a hard RT system, it is safer for a task to produce less accurate result on time, rather than producing the accurate result, late. The motive is to employ IC with the proposed RT DEVS approach in order to have a formal platform for designing hard RT systems. The objective is to address the above challenges in the proposed DEVS-based RT design scheme, without complicating the formalism or adding extra processing burden and maintaining the backward compatibility in order to reuse previous models.

1.1 Contributions

The main contributions in this dissertation are to propose a DEVS-based simulation-driven development methodology for RT and embedded systems and also a hard RT system design scheme by integrating the proposed approach with IC technique. One of the key contributions regarding these research objectives is a new M&S-driven approach referred to as **DEVSRT (Discrete-Event Systems Specifications in Real-Time)**, a domain extension to DEVS theory for embedded real-time application development. The DEVSRT takes advantage of well-defined M&S properties and constructs of DEVS to design and interface embedded systems with the hardware and the plant under study. DEVSRT approach includes the following contributions:

- The notion of deadline is added to the DEVS formalism making it appropriate for real-time system modeling and design. Based on DEVS computational properties, a set of assumptions are defined to be used in designing a real-time system. DEVSRT uses DEVS formal outputs as output signals of the real-time system, therefore a relative deadline is associated with each output produced at the end of each state.
- An efficient interfacing mechanism is added to the DEVS theory. This will satisfy the following major motivations of this research: 1) **Model continuity** from simulation stage up to embedding the models in the target hardware. 2) The entire system is designed in a **hardware-software co-design** approach, in which the models represent different

hardware and software components and are tested together as an integrated DEVS model. 3) Provides **hardware-in-the-loop simulation** platform where some of the models act as the simulated plant components (in which the driver interfaces provide the electrical emulator signals) and are tested with the embedded system (controller) model to be deployed on the hardware.

- A generic lightweight interface for message transfers between DEVS models running on different DEVS-based tools is presented. This provides a basis for component-oriented collaborative modeling and simulation with other DEVS-based tools, allowing reusing other models or using special services offered by other tools.
- The Embedded CD++ (E-CD++) tool [Yu07a, Yu07b] is extended as a software environment to implement the proposed DEVSRT framework for the formal development of embedded real-time applications. The new version of E-CD++ is implemented on a real-time kernel, incorporating real-time tasking services. Many new object-oriented entities and capabilities are added to this framework in order to support the RT functionalities, driver interfaces, and other features of DEVSRT.
- An Integrated Development Environment (IDE) containing embedded functionalities and graphical model designer capability is provided, which permits rapid design and deployment of the models.

Finally, this approach has been used to develop various real-time embedded systems on a variety of hardware platforms (such as FPGAs, embedded boards, and robotic devices), and collaborative models. A test case using a robotic model application is presented here and the development process and results are discussed.

The second contribution of this dissertation includes: integrating the M&S-based approach proposed in DEVSRT with the (Imprecise Computations) IC technique to build a more reliable design and execution platform for hard real-time systems.

- The Imprecise DEVS (I-DEVS) formalism is proposed, providing flexibility to the user by separating the behavior of the system to mandatory and optional, to achieve a more reliable RT task scheduling from the processor.
- A detailed (model-independent) task model of DEVS formalism is presented, which identifies processing tasks being executed in a real-time DEVS-based system. This proposal is then used to develop RT IC-based scheduling algorithms.
- An early reaction algorithm to the overrun conditions in DEVS-based hard real-time systems is proposed, in which the system can act early enough to save the critical tasks from lateness.
- The I-DEVS M&S framework is implemented on E-CD++, providing a development platform for imprecise modeling and execution using DEVS formalism.

1.2 Research Publications

Some of the research results of this dissertation are published so far. The publications are categorized in four categories. 1) Publications related to DEVSRT and models developed using this framework, 2) publications regarding I-DEVS formalism, 3) publications regarding the RT collaborative modeling scheme, and 4) other publications concerning DEVS-based modeling and simulation.

The following publications are related to the DEVSRT and DEVS-based embedded system design:

- Moallemi, M., M. Alcaraz, and G. Wainer, “ECD++ A DEVS based Real-Time Simulator for Embedded Systems”, Poster in proceedings of Spring Simulation Conference, Ottawa, Canada, 2008. This paper presents the basic real-time simulation and embedded functionalities added to E-CD++ which was later used as a basis for driver function addition. The four embedded functionalities added to the Eclipse IDE of E-CD++ are also discussed in this paper.

- Holman, K., J. Kuzub, M. Moallemi, G. A. Wainer, “Cable-Anchor Robot Implementation using Embedded CD++”, Poster in proceedings of SIMUTools Conference, Rome, Italy, 2009. This poster paper presents a real-time model for a cable-anchor robot, developed using the DEVSRT framework. The model specification and implementation details are presented.
- Moallemi, M., and G. A. Wainer, “A System-On-Chip FPGA Implementation of Embedded CD++”, Proceedings of Spring Simulation Conference, San Diego, CA, USA, 2009. This paper proposes an FPGA-based implementation of the so far DEVSRT formalism. The RT approach was implemented on a Virtex2pro Xilinx FPGA board, on an embedded Linux environment. Different models have been tested on this platform.
- Moallemi, M., and Gabriel Wainer, “A Simplified Real-Time Embedded DEVS Approach Towards Embedded and Control Design”, Poster in proceedings of Winter Simulation Conference, Austin, USA, 2010. This poster paper presents some of the later details and refinements of DEVSRT framework, in which the theory is more revealed.
- Moallemi, M., and G. A. Wainer, “Designing an Interface for Real-Time and Embedded DEVS”, Proceedings of Spring Simulation Conference, DEVS Symposium, Orlando, Florida, USA, 2010. In this paper the details of driver interface functions and implementation of DEVSRT on E-CD++ are discussed. The real-time platform, where the E-CD++ is implemented and the DEVS model integration with hardware is presented.
- Moallemi, M., D. A. Tall, G. A. Wainer, and A. Awad, “Application of RT-DEVS in Military”, Proceedings of Spring Simulation Conference, MMS Symposium, Orlando, Florida, USA, 2010. This paper presents an RT and embedded DEVSRT model for a reconnaissance tank. The model is designed as DEVS model, tested in simulation mode and then deployed on the hardware following the DEVSRT framework.
- Sadeghi, F. R., G. Wainer, and M. Moallemi “Modeling and Controlling a Robotic Arm with E-CD++”, Poster in proceedings of Summer Simulation Conference, Ottawa, ON, Canada, 2010. In this paper, a model of an advanced robotic arm is presented, following

the proposed DEVSRT approach. The robotic arm is used in medical robotics, remote surgery, and medical simulation.

The following publications are related to the collaborative modeling scheme:

- Moallemi, M., R. Castro, F. Bergero, and G. A. Wainer, “Component-Oriented Interoperation of Real-Time DEVS Engines”, Proceedings of Spring Simulation Conference, ANSS Symposium, Boston, MA, USA, 2011. This paper presents the main idea of collaborative modeling and model execution based on component-oriented nature of DEVS, using the interfacing and RT features of DEVSRT. The pros and cons of this approach are discussed and some limitations that must be observed are briefed.
- Moallemi, M., S. Jafer, A. S. Ahmed, and G. Wainer “Interfacing DEVS and Visualization Models for Emergency Management”, Proceedings of Spring Simulation Conference, Work In Progress of the DEVS Symposium, Boston, MA, USA, 2011. This paper introduces a method to integrate Cell-DEVS [Wai02a] models with DEVS-based robotic agents and an advanced immersive visualization environment for Emergency Management, using the proposed collaborative approach. The emergency is handled by an autonomous robot controlled by a real-time DEVS model. The model controlling the robot interacts with a simulation for emergencies, receiving real-time data about its location on a cell space. The immersive environment is used to visualize the emergency and its management.
- Ahmed, A. S., M. Moallemi, G. Wainer, and S. Mahmoud, “Cell-DEVS & 3D Real-Time Visual Simulation to Support Combat”, Proceedings of Summer Simulation Conference (SCSC'11), Netherland, 2011 (**Runner up Best Paper Award**). This paper presents the design and development of a collaborative 3D real-time visual Cellular Agent model (VCELL). VCELL is used for simulating land combat and is collaboratively modeled comprising a Cell-DEVS agent model and an advanced visual immersive simulation environment.

The following publications are related to the I-DEVS approach:

- Moallemi, M., and G. A. Wainer, “I-DEVS: Imprecise Real-Time and Embedded DEVS Modeling”, Proceedings of Spring Simulation Conference, DEVS Symposium, Boston, USA, 2011 (**Best Paper Award**). This paper proposes the initial I-DEVS formalism, and the RT task-scheduling algorithm based on IC technique. This approach combines the dynamic advantages of the imprecise computation technique with the rigor of a formal modeling methodology. A synthetic robotic model is developed and the results of the execution of this model in different scenarios are compared and discussed.

The following publications are general DEVS-based M&S researches:

- Moallemi, M., and G. Wainer, “Design of Persian Tapestry in CD++”, Poster in proceedings of Spring Simulation Multi-conference, Ottawa, Canada, 2008. This paper presents a cellular automata [Neu66] model of a Persian carpet using Cell-DEVS formalism and implemented on CD++ tool. The model produces different fascinating tapestry shapes which can be used to weave real tapestries.
- Moallemi, M., A. Arya, and G. Wainer, “Simulation of Three Dimensional Elevator System Using Cell-DEVS Formalism”, Proceedings of Spring Simulation Conference, ANSS Symposium, Orlando, USA, 2010. This paper proposes a cellular model of a 3D elevator system used in high-rise buildings. The paper investigates different strategies to be considered in the design of the system and routing the elevator cars.

1.3 Organization

The rest of this dissertation is organized as follows: Chapter 2 presents some background information on the general aspects of M&S and then discusses DEVS formalism and different variants of it. Classical DEVS, Parallel DEVS, and RT-DEVS are reviewed and the abstract simulation algorithm of the parallel DEVS (which is the basis for DEVSRT) is presented. Chapter 3 describes several related and similar RT and embedded system development approaches and tools, and discusses the pros and cons of these approaches. The need for an M&S-based approach supporting model continuity and other necessary features for such systems

is justified in this chapter. Later, the DEVSRT formalism, the details of DEVS task model, the model interfacing mechanism, and the details of implementation of the scheme on E-CD++ are presented. An example of a robotic controller model and the details of design, implementation and deployment of the model are presented, where the simulation results are compared with the actual system performance. Chapter 4 presents practical examples of the extended applications of DEVSRT in collaborative modeling, and integration of RT models with virtual reality environments. The messaging scheme, the interfacing mechanism, the limitations and finally the models' details are provided. Chapter 5 proposes the I-DEVS approach, where the RT tasks working in the context of a DEVS-based system are extracted and then an RT scheduling algorithm, confined in the DEVS atomic component level is proposed. The results of the execution of a synthetic model using precise and imprecise modes are compared and the effectiveness of this approach is shown. Chapter 6 concludes the dissertation and proposes some future research directions for future researchers and students in this subject.

Chapter 2: Background

As discussed earlier, the need for high quality software with no defect for real-time (RT) and embedded applications has evolved techniques in which system specifications are expressed in clean mathematical basis. M&S-based approaches provide various advantages however, they are not as robust as formal methods, and the lack of a formal foundation for M&S poses difficulties when trying to prove properties about the embedded systems modeled. Thereby, formal M&S-based approaches are good alternatives as they provide a mathematical-driven system specification framework, suitable for the design and development of such systems. Another important challenge is the issue of consistency and traceability from the design stage to the deployment [Bou05]. There are no well-established techniques in M&S-based design schemes to bridge the gap between the modeling and the hardware deployment phases, nor techniques for mapping the model behavior to an RT task system to adopt task scheduling algorithms in real-time operating systems. Because of these drawbacks in using M&S-based design, often M&S artifacts are abandoned and not used for the development of the actual embedded system, resulting in extra development costs [Wai09].

This chapter presents the DEVS M&S framework, different variants of the DEVS formalism, its abstract simulation algorithm, and details of model development on CD++ software. Subsequently, the available M&S-based methodologies and tools for RT and embedded system development are evaluated and the challenges in each of them are explored.

2.1 Modeling and Simulation Concepts in DEVS

The theory of discrete event modeling and simulation is more or less a recent innovation tied to the advancement of computer systems. Based on Zeigler et al. [Zei00], the Discrete-Event System Specification (DEVS) theory observes the system of interest (source system) as a set of behavioral data operating in the context of an experimental frame (EF) with a set of conditions. A model of this system provides an abstract representation of the system under study, by means of mathematical equations and instructions. Figure 2.1 illustrates this M&S framework with its entities and relations. The entities are source system, experimental frame, model, and simulation, and there are two types of relations: modeling relation, and simulation relation. The system under study is illustrated as the source system, working in the general framework of the EF, which is of interest to the modeler. The model represents the source system, its structure and behavior in an abstract way, including part of the reality of the source system and its working conditions.

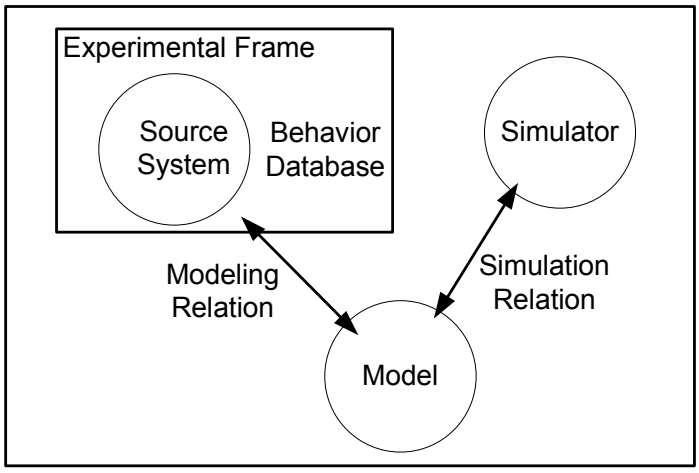

Figure 2.1: Entities in a DEVS-based M&S framework [Zei00]

The model is expressed as a set of instructions, rules, mathematical equations, or constraints that are used to approximate the I/O trajectories of the source system. Consequently, the

simulation is a computational implementation of an available model to execute the model and extract these I/O trajectories.

The relations defined in Zeigler's M&S framework envision the relation between EF and the model and between the model and the simulator. The modeling relation affects the accuracy of the behavior generated by the model compared to the behavior of the actual source system, while the simulation relation is concerned with the accuracy of the executed simulation results compared to the model that has been defined. The operational details of the simulator (e.g. software, hardware …) affect the precision and limitations of the model. Different verification techniques try to detect discrepancies between the model definition and the source system. On the other hand, validation tests concern with inconsistencies between the simulation results and the source system [Sar87].

This separation of data (model) and control (simulation) enables the modeler to confine the efforts to model design, while using existing simulators to execute the model. On the other hand, the use of formal techniques to describe the model provides mathematical proofs for the model. The other advantage of this method is the ability to execute the same model on different simulators implementing the same formalism, providing a potential for portability and interoperability of the simulators and benefiting from the unique features of different simulators. This scheme allows the model and simulator to evolve separately and maintain consistency. This technique can be presented as a layered approach for executing the model using computer-based simulation [Wai09]. Figure 2.2 illustrates the M&S layers in a DEVS-based simulation system.

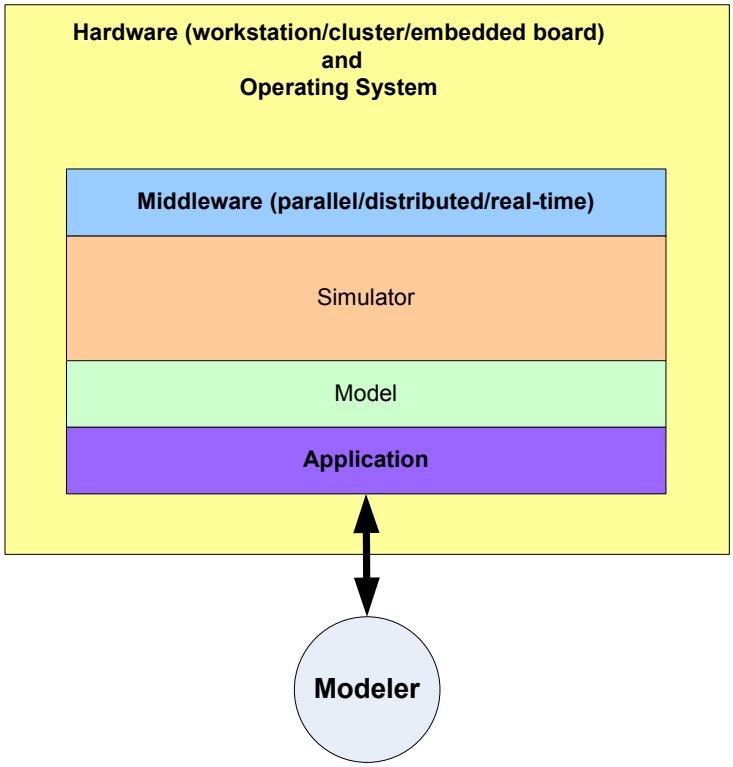

Figure 2.2: M&S layers in a DEVS-based system (modified from [Wai09])

The modeler defines the model using a dedicated computer application providing specific tools for expressing a DEVS model. After that, the simulator executes the model. The simulator can incorporate different middleware technologies to hide the execution details from the modeler. Middleware is the bridge between the simulator and the hardware, providing special services regarding the details of execution of a model. The model is executed on a hardware platform with a specific operating system or as an embedded application on an embedded board with no operating system[1].

[1] Not to be confused with an embedded control system. An “embedded simulation” executes a simulation model on an embedded board and outputs the results of the simulation.

This dissertation concerns with designing a new real-time simulation approach capable of being transformed to an RT system that executes a model as a control algorithm in the framework of an embedded system.

2.2 Classical DEVS Formalism

As discussed in the previous section, the modeling and simulation aspects of a DEVS system are separated in order to modularize and formulate the design of a model, based on the requirements of the source system. To this end, DEVS has been proposed as a sound formal framework for modeling generic dynamic systems and includes hierarchical, modular and component-oriented structure and formal specifications for defining structure and behavior of a discrete event model [Zie00]. A DEVS model is comprised of structural (Coupled) and behavioral (Atomic) components, in which the coupled component maintains the hierarchical structure of the system, while each atomic component represents a behavior of a part of the system. The atomic component is the basic building block of the system which is composed of I/O ports and a finite state timed automaton representing the behavior of the model. An input to the atomic component via an input port triggers a state transition (referred to as “external transition”), and in contrast the state transition (referred to as “internal transition”) at the end of the time-delay of each state leads to an output generation through an output port. This dynamic behavior is represented using the following formal notations [Zei00]:

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

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

S is the set of states;

$\delta_{ext}: Q \times X \rightarrow S$, is the external transition function

Where Q is the total state set of $M = \{(s, e) \mid s \in S \text{ and } 0 \leq e \leq ta(s)\}$

$\delta_{int}: S \rightarrow S$, is the internal transition function

$\lambda: S \rightarrow Y$, is the output function

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

An atomic component AM is affected by external input events X which in turn generates output events Y. The internal transition function δ_{int} and the external transition function δ_{ext} compute the next state of the model. If an external event arrives at elapsed time e which is less than or equal to ta(s) specified by the time advance function ta, a new state s′ is computed by the external transition function δ_{ext}. Then, a new ta(s′) is computed, and the elapsed time e is set to zero. Otherwise, a new state s′ is computed by the internal transition function δ_{int}. In the case of an internal event, the output specified by the output function λ is produced based on the state s and a new ta(s′) is computed, and the elapsed time e is set to zero.

Figure 2.3 illustrates the state transition of an atomic component. An atomic component is in state s for a specified time ta(s). If the atomic component passes this time without interruption it will produce an output y at the end of this time and change state based on its δ_{int} function (internal transition) and continues the same behavior. However, if it receives an input x during its ta(s) time, it changes its state which is determined by its δ_{ext} function and does not produce an output (external transition).

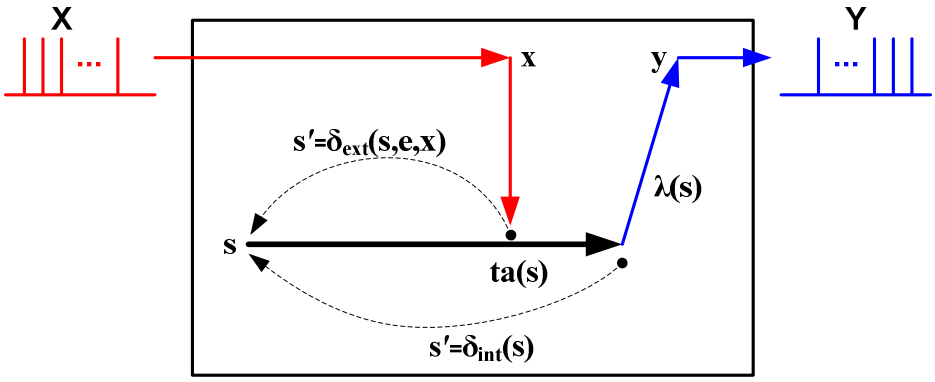

Figure 2.3: DEVS atomic component state transition sequence (modified from [Wai09])

A coupled model connects the basic models together in order to form a new model. This model can itself be employed as a component in a larger coupled model, thereby allowing the hierarchical construction of complex models. The coupled model is defined as [Zei00]:

$CM = <X_{self}, Y_{self}, D, \{M_i \mid i \in D\}, \{I_i\}, \{Z_{i,j}\}, \textit{Select}>$, where:

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names;

for each i in D , M_i is a component;

for each i in $D \cup \{self\}$, I_i is the influencees of i;

for each j in I_i, $Z_{i,j}$ is a function, the i-to-j output translation;

SELECT: 2M − Á → M, the tie-breaking selector;

The structure is subject to the constraints that for each i in D,

$M_i =< X_i, S_i, Y_i, \delta_i, \lambda_i, ta_i >$

I_i is a subset of $D \cup \{self\}$, i is not in I_i ;

$Z_{self,j} : X_{self} \rightarrow X_j$;

$Z_{i,self} : Y_i \rightarrow Y_{self}$;

$Z_{i,j} : Y_i \rightarrow X_j$;

SELECT: subset of $D \rightarrow D$, such that for any non-empty subset E, $SELECT(E) \in E$.

A coupled model CM consists of components $\{M_i\}$, which are atomic components and/or coupled models. The influencees $\{I_i\}$ and the i-to-j output translation $\{Z_{i,j}\}$ define three types of coupling specifications. The external input coupling connects the input events of the coupled model itself to one or more of the input events of its components. The external output coupling connects the output events of the components to the output events of the coupled model itself. The internal coupling connects the output events of the components to the input events of other components. The SELECT function is used to order the processing of the simultaneous events for sequential events. Thus, all the events with the same time in the system can be ordered with this function.

Figure 2.4 shows a hierarchical DEVS model. This model is composed of three atomic components (Generator, Buffer and Processor) and two coupled models: the top-most coupled model (GEN-BUF-PROC) that contains generator atomic component and BUF-PROC coupled

model, which itself includes two atomic components: BUF and PROC. The port connections are also visible in the figure. For example, the output port “out” of atomic component PROC is connected to the “done” input port of BUF atomic component within the BUF-PROC coupled model and also is connected to the output port of its parent coupled model which connects this output to the top-most model output port.

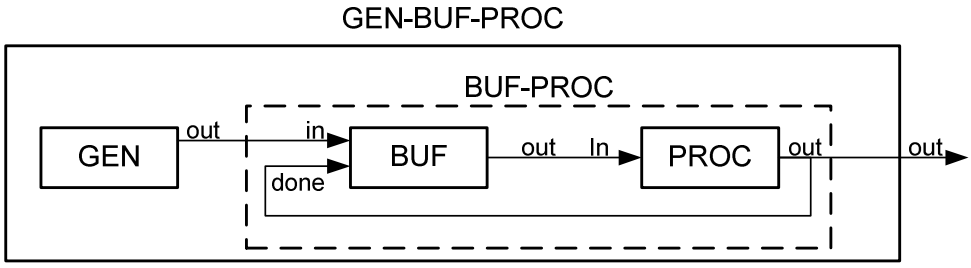

Figure 2.4: Coupled DEVS model example [Wai09].

2.3 Parallel DEVS Formalism

The *Select* function in classical DEVS formalism handles simultaneous events by serializing their occurrences based on the modeler’s preference, which reflects the closest sequence happening in the real system. Let’s consider a scenario in which multiple imminent components (a component which is supposed to execute an internal transition) exist at a certain time in a coupled component. In this case, different order of the execution of the transitions in these components will produce completely different behaviors in the entire system and will affect the results, subsequently. To manage these ambiguities, the modeler specifies the imminent component that has the priority to execute its transition, using the *Select* function and based on the results, the rest of the components will be served in the next simulation cycle.

This technique however proposes a number of limitations. The arranged order of the events might not exactly represent the reality in the source system. The circumstances might change dynamically, while this technique only proposes a fixed ordering in all instances. Besides, this method does not provide a solution for potential parallel execution of the events. To address

these issues Chow and Zeigler proposed the **Parallel DEVS** (P-DEVS) formalism [Cho94] as a solution.

The P-DEVS formalism handles simultaneous events inside the atomic component by introducing a new transition function referred to as “confluent function”. It also introduces an input bag in the atomic component, where simultaneous inputs are stored there and are serviced afterwards. This allows the external function to handle multiple simultaneous inputs in one function call. On the other hand simultaneous internal and external events are handled in the confluent function by the modeler, which can be dynamically adapted to the circumstances. This technique also provides a potential for parallelism in the execution of the model by allowing all the atomic imminent components to be activated at the same time [Zie03].

The P-DEVS atomic component has the following structure [Cho94]:

$AM = < X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$, where:

$X_M, Y_M, S, \delta_{ext}, \delta_{int}, \lambda$ and ta are the same as DEVS atomic component specifications.

δ_{con}: $Q \times X_M^b \rightarrow S$ is the confluent transition function;

The semantics of the P-DEVS model definitions are as follows. At any given time, a basic model is in a state s. In the absence of external events, it will remain in that state for a period of time determined by ta(s). When an internal transition takes place, the system outputs the value $\lambda(s)$, and transitions to state $\delta_{int}(s)$. If one or more external events $E = \{x_1 \ldots x_n / x \in X_M\}$ occurs before ta(s) expires, (i.e., when the system is in the state (s, e) with $e \leq ta(s)$) the new state will be given by $\delta_{ext}(s, e, E)$. Suppose that an external and an internal transition collide, (i.e., an external event E arrives when e = ta(s)) the new system’s state could either be given by $\delta_{ext}(\delta_{int}(s), e, E)$ or $\delta_{int}(\delta_{ext}(s, e, E))$. The modeler can define the most appropriate behavior with the δ_{con} function. As a result, the new system’s state will be the one defined by $\delta_{con}(s, E)$.

A P-DEVS coupled model (CM) is defined the same as DEVS model except that there is no tie breaking function (SELECT), as this problem is solved within the atomic component using δ_{con} function.

A) Abstract Simulation Algorithm

As mentioned in section 2.1, DEVS M&S theory separates the modeling entity from the simulation entity, in order to increase the abstraction level and dynamism of the framework. In addition to providing rigorous model definition formalism, it also provides an abstract simulation mechanism to understand and execute a DEVS-based model. The processors associated with atomic and coupled components in the abstract simulation mechanism, are referred to as *simulators* and *coordinators*, respectively. Thus, the processor hierarchy (or control hierarchy) is composed of coordinators as middle nodes and simulators as the leaves. There is a top-most coordinator referred to as *Root Coordinator* (RC) (reflecting the top-most coupled component) which initiates each simulation phase by sending the following messages to the simulators: (q, t) representing an *input message* that carries an input value from external environment and the time stamp of the message. (@, t) also referred to as *collect message* carrying a signal to the simulator in order to generate an output. (*, t) referred to as *internal message* carrying a wake up call to an imminent simulator. The simulator in response replies with the following messages: (done, t) referred to as *done message* which is produced in response to input and internal messages after invoking the transition functions, carrying the duration of their new state (ta(s)). (y, t) referred to as *output message*, which is produced in response to a collect message, carrying the output value.

An en-route coordinator is responsible for converting output messages to input messages in case of an internal coupling. It is also responsible for sending the smallest time of internal event (also referred to as *next change* (t_N)) among its components whenever it is forwarding a done message up to the RC. The t_N time is the relative time from now up to the time when the imminent children of a coordinator must be invoked in order to perform an internal transition. The *last change* time (t_L) is the relative time from the last activity in a component to the current time.

The following pseudo code snippet represents collect message handling algorithm in a P-DEVS simulator [Cho94].

```
1. when receive (@, t):
```

```
2. if (t = t_N) then
3.     y = λ (s)
4.     send (y, t) to the parent coordinator
5.     send (done, t) to the parent coordinator
6. end
7. else if
8.     error
9. end when
```

The following pseudo code snippet represents input message handling algorithm in P-DEVS simulator [Cho94].

```
1. when receive (q, t):
2. lock the bag
3. Add event q to the bag
4. unlock the bag
5. send (done, t) to the parent coordinator
6. end when
```

In P-DEVS, the receipt of input message does not trigger the external transition function. Instead, the input is inserted in the bag, allowing for processing simultaneous inputs stored in the bag, when an internal message is received. Thus, an internal message must always accompany an input message.

The following pseudo code snippet represents internal message handling algorithm in P-DEVS simulator [Cho94].

```
1. when receive (*, t):
2. if (t_L ≤ t < t_N) and bag is not empty
3.     e = t  - t_L
```

4. $s = \delta_{ext}(s, e, bag)$
5. empty bag
6. $t_L = t$
7. $t_N = t_L + ta(s)$
8. end if
9. else if ($t = t_N$) and bag is empty
10. $s = \delta_{int}(s)$
11. $t_L = t$
12. $t_N = t_L + ta(s)$
13. end if
14. else if ($t = t_N$) and bag is not empty
15. $s = \delta con(s, bag)$
16. empty bag
17. $t_L = t$
18. $t_N = t_L + ta(s)$
19. end if
20. else if ($t > t_N$ or $t < t_L$)
21. error
22. end if
23. send (done, t_N) to parent coordinator
24. end when

The internal message will produce three circumstances in an atomic component based on the current time and input bag conditions. Line 2 refers to a case in which, the internal message is received sometime before the end of current state's life time and the input bag is not empty. This means there are inputs to be serviced; hence the external transition is invoked. Line 9 refers to the case when the internal message is received at the end of the lifetime of the current state,

while the input bag is empty, indicating an internal transition. Line 14 shows the case when internal message is received at the end of the state and there are inputs to be served. Therefore, confluent function must be called to handle the collision of external and internal transitions.

2.4 Real-Time DEVS Formalism

Hong et al. [Hon97] proposed a real-time version of DEVS formalism as an extension of the classical DEVS for real-time systems simulation. An atomic component in RT-DEVS formalism (RTAM) is defined as:

RTAM=<X, S, Y, δ_{ext}, δ_{int}, λ, ta, ti, ψ, A >, where:

X, S, Y, δ_{int}, λ and ta are the same as original DEVS.

δ_{ext}: Q × X→S, an external transition function, where Q is the total state set of M= {(s, e)|s∈ S and $0 \leq e \leq ti(s)|_{max}$}

ti: a time interval function,

ψ: an activity mapping function,

A: a set of activities,

With constraints:

ψ: S→A

ti: S→ $R^{+}_{0,\infty} \times R^{+}_{0,\infty}$,

Where $ti(s)|_{min} \leq t(a) \leq ti(s)|_{max}$, $ti(s)|_{min} \leq ta(s) \leq ti(s)|_{max}$, s∈ S, a = ψ (s) ∈ A and t(a) is the execution time of an activity a.

A= {a| t(a) ∈ $R^{+}_{0,\infty}$, a∉ {X?, Y!, S=}}

Where: X? is the action of receiving data from X, Y! is the action of sending data from Y and S= is the action of modifying a state in S.

In RT-DEVS an activity mapping function ψ and an activity set A are defined to advance virtual time with an executable activity associated with an event. The regular ta time advance function only verifies the correctness of activity mapping time constraints and compensates time discrepancy problems. The time bound of each activity is specified by ti function.

A coupled model within the RT-DEVS formalism is defined the same way as in the original DEVS formalism with an exception. The exception is that there is no *SELECT* function in RT-DEVS, which has been defined in the DEVS formalism to break ties for simultaneous events scheduling. The authors justify this by claiming that in a real-time simulation environment simultaneous events do not occur. In real-time simulation with one processor, only one event at a time can be physically processed even if more than one event occurred from the external environment.

2.5 DEVS Simulation in CD++

CD++ [Wai09] is an open-source simulation software that implements the DEVS abstract simulation technique. In CD++, simulators and coordinators progress through the simulation by exchanging messages as described by the DEVS abstract simulation mechanism. CD++ benefits from object-oriented design, allowing the developer to make use of powerful object-oriented techniques in integrating simulation entity with modeling entities developed by the modeler. The rest of this section reviews CD++ software architecture and how it implements the DEVS simulator.

A) CD++ Software Architecture

CD++ is designed as an object-oriented simulation engine, modularized as a group of components that have well-defined behaviors and have relatively independent functionalities. CD++ architecture consists of the following major components [Yu07b]: *Main Simulator*, *DEVS Modeling Subsystem*, *Simulation Subsystem*, and *Messaging Subsystem*.

Main Simulator manages the overall aspects of the simulation and is the first object that is created when the simulator starts. In general, the following tasks are performed by Main Simulator:

- Atomic component classes defined by modeler as C++ objects derived from the Atomic class are registered in a list. These atomic models will be instantiated during the model loading process, which is the next step performed by the Main Simulator.
- The DEVS model hierarchy is constructed by parsing the modeler-defined *model-file* in which the DEVS components and their couplings are declared (e.g., atomic and coupled models, ports, links, states durations, etc.). During this phase, two hierarchical C++ tree objects are constructed, one representing the modeling (Atomic and Coupled) components (Modeling Subsystem) and the other one representing the simulation (Simulator and Coordinator) processors (Simulation Subsystem). For each atomic or coupled object a Simulator or Coordinator object is created, respectively complying with DEVS abstract simulator mechanism discussed in section 2.3A).
- After this, the external events are loaded from the *event-file* (in case there is one), and the Root Coordinator (RC) corresponding to the Top coupled component is created. RC is responsible for starting and controlling the simulation cycles and advancing the time based on the order of the events (in a virtual time simulator).

The *Simulation Subsystem* consists of *Simulators*, *Coordinators*, and the *Processor Manager*. The control messages defined in abstract simulator are transferred among these objects, while the behavior functions (transitions, time advance, and output function) are implemented in the modeling subsystem. RC generates the very first message in the simulation, which triggers other processors to receive and send messages. Coordinators forward the messages to their children or parents and also maintain a list of their imminent children. Simulators have a pointer to their corresponding Atomic object instances and invoke user-implemented behavior functions after receiving the appropriate message. RC advances the simulation time and stops the simulation cycle when all models become passive (i.e. all atomic components are in a state with lifetime of infinity) and there is no external events left to process, or when the user-specified simulation end time arrives.

The Modeling Subsystem maintains the model hierarchy information presented by the user in model file. The subsystem is composed of coupled and Atomic component objects, Input and Output port objects and the *Model Manager* which keeps a hashing table of the model components and inter-couplings.

The Messaging Subsystem consists of the *Message Manager* and various *Message* classes. *Messages Manager* is responsible for delivering messages transferred among the Coordinators and Simulators. The incoming messages are first buffered into the *Message Queue* and are processed by the *Messages Manager* in FIFO order. The Message fields are *sender, receiver, time-stamp, value,* and *port.*

B) DEVS Model Definition in CD++

Based on the software structure described above, the modeler has to provide the following information to build a DEVS model in CD++: 1) Model specifications, 2) Events, 3) Atomic component behavior.

Model specifications are defined in the *model-file* in special format defined in [Wai02b]. The model-file contains the components (Atomic and Coupled) in a top to bottom order, in which the top-most coupled component is declared first. For each Coupled component, its internal components, I/O ports, and links (EIC, EOC, and IC) are declared. For each atomic component its state durations can be defined in the model-file, in order to easily modify them during various execution scenarios without recompiling the source code. The Main Simulator Subsystem automatically instantiates Atomic and Coupled objects for each component based on the order defined in the model-file.

Events are defined in event-file with a specific format, containing the event time, value, and input port. The Main Simulator stores the events in an event queue at the beginning of the execution and RC injects the event to their associated input port at the specified time.

Atomic components in CD++ are overridden by the modeler as sub-classes of the Atomic class. The modeler implements the desired behavior by programming the following four

functions: *init function*, *external function*, *internal function*, and *output function*. These functions are invoked using polymorphic techniques inside the Simulator class corresponding to that Atomic component. The states are declared in the atomic component classes using an enumerator type, and state durations are set by calling the *holdIn*, or *passivate* functions. Finally the Atomic component object pointers are handed to the Main Simulator in its modeler-overridden *registerAtomics* function.

2.6 Modeling and Simulation-based Approaches

Various modeling methodologies have been introduced in literature, concerning the design and development of real-time and embedded software systems. A typical M&S-based approach towards this end consists of the following steps: 1) specification (requirements and constraints); 2) modeling; 3) simulation/verification; 4) model mapping to hardware/software components; 5) prototyping and implementation. Among the model-based approaches, the UML-RT (the Unified Modeling Language for Real-Time) [Sel01] is an extension of UML modeling language which provides especial aspects for designing real-time systems. UML and UMLRT have been used in applying model-based design, verification, and performance analysis of different systems (see e.g. [Cor01 and D'Am05]). A comparison between DEVS and UML-RT [Hua04] shows that, although features such as time, scheduling and performance are coded using UML constructions (i.e., not formally defined). Instead, DEVS provides sound syntax/semantics for structure, behavior, time representation and composition, which lend themselves to well-defined computations. DEVS, however, is not intended for software design and development, and "it is key to support the transformation of simulation models to their software model counterparts and their complementary roles in handling modeling and computational complexity of embedded systems" [Hua04]. This research aims at overcoming these issues by introducing the model continuity and other modifications to the DEVS M&S framework.

Among other model-based approaches, the BIP (Behavior, Interaction, Priority) methodology is a framework for heterogeneous component-based modeling of real-time systems introduced in

[Bas06]. Components are obtained as the superposition of three layers: Behavior; specified as a set of transitions, Interactions; between transitions of the behavior, and Priorities; used to choose amongst possible interactions. BIP does not support simulation-driven approach which plays a key role in performances analysis and reliability of the software. The issue of model continuity is handled using code generation tool, which lacks a direct traceability relation between the model and the final software architecture.

The following research efforts apply simulation-based design which is more closely related to this work. ECSL (Embedded Control Systems Language) is a tool-suite that supports software development for distributed embedded controllers [Bal06]. ECSL offers a graphical modeling language built using the Generic Modeling Environment (GME) [Led01], an open-source meta-programmable domain-specific design environment. It was designed in the context of embedded automotive systems with capabilities such as requirements specification, verification, mapping on to a distributed platform, scheduling and performance analysis.

Ptolemy II [Eke03] is a structured and hierarchical method for modeling heterogeneous systems using a specific model of computation that covers the flow of data and control. SystemC [Gro02] and Esterel [Bou91] are system description languages that can be used for generating simulatable and executable models. They share some features and also have their unique characteristics, and some two-way component mapping can be performed between the languages.

Matlab/Simulink® is a commercial tool provided by Mathworks [Mat11] for modeling and simulating embedded systems that also offers graphical interface for visual construction and integration of hardware blocks. Simulink® can be integrated with different other tools provided by Mathworks, such as Stateflow®, Simulink Coder®, and Embedded Coder® for event-based modeling, physical modeling, and code generation. Simulink is mainly used for simulating real-time systems, while the accompanied code generation tool produces C/C++ code for embedded processors. Nevertheless, the generated code has limited usage, does not support all the functionalities of the Simulink blocks and still needs verification.

None of the above-mentioned modeling approaches provide direct model continuity, whereas DEVSRT allows for straight use of the simulation models as the final target software. The other advantage is the straightforward hardware-software co-design capability [43] (i.e. co-specification, co-synthesis, co-simulation and co-refinement) in a more abstract level as well as hybrid testing of simulation models with real hardware. The use of DEVS simplifies the transformation of the models from various other formal methods, supporting heterogeneous systems design, implementation, and reuse, which is necessary in embedded system development. DEVS, as a simulation methodology, not only allows for simulation-driven software development but also supports M&S of an entire system and its surrounding environment. This allows for verification of the software in a simulated environment with changing conditions.

2.7 DEVS-Based Approaches

In [Sha07] the authors introduce the FDS-DEVS (Flexible Dynamic Structure DEVS) algorithm based on [Bar97] dynamic structure system modeling approach. FDS-DEVS enables adapting a simulated system's organization to the dynamically changing internal/external environment while the system execution is in progress. The authors propose an MDA (Model Driven Architecture) technology applied to real-time DEVS experimental environment, which is greatly benefited from dynamic structure capabilities. This methodology allows building real-time DEVS-based simulation models capable of changing their components dynamically, based on real-world systems. This enhances the capabilities of DEVS-based approaches in building reliable and adaptive real-time systems as they can respond to the changing contexts or recover from errors automatically.

In [Sar01a] an application of the DEVS/DOC (Discrete Event System Specification/Distributed Object Computing) co-design methodology is presented. In this case study model, the architectural and scalability aspects of a Mission Training and Rehearsal System (MTRS) is analyzed and implemented, in which "it has to be accounted concurrently for

hardware and software requirements, given high demands for network bandwidth, computing resources, and complexity of software applications". The authors showed the advantage of the DEVS/DOC approach applied to Software/Hardware design from architectural, behavioral, and performance viewpoints. In this scenario the system demands a systematic approach to transition from high-level system design to low-level component design. DEVS/DOC can help detect architectural errors before they lead to lower level system failure, and offer a suitable solution for high level system specification, by introducing a modeling layer on top of fine grained DEVS modeling constructs.

In [Hu01], a DEVS-based RT system has been implemented on a TINI Chip which has limited memory and processing ability. A set of well-defined DEVS Interfaces made it possible to define a just-as-needed RT environment and run on the chip efficiently. Finally a case study model has been successfully run on the chip.

In [Hua06] a modeling approach for semiconductor manufacturing supply-chain systems in a hybrid DEVS/MPC (Model Predictive Control) test-bed that supports experimentations for DEVS and MPC models using $KIB_{DEVS/MPC}$ (Knowledge Interchange Broker) is proposed. This test-bed supports detailed analysis and design of interactions between discrete processes and tactical controller. In this work, the DEVS model captures complex dynamics of semiconductor manufacturing processes whereas the MPC model is responsible for tactical control.

In [Hu05] the authors show how an M&S environment based on the DEVS formalism can support model continuity in the design of dynamic distributed real-time systems. The authors prove that the discontinuity between implementation artifacts and analysis, design, and modeling artifacts is a common deficiency of most design methods. The authors restrict model continuity to the models that implement the system's real-time control and dynamic reconfiguration, and emphasize model continuity during the entire process of software development, where the control models can be designed, analyzed, and tested by simulation methods, and then smoothly transitioned from simulation to distributed execution. The proposed methodology supports model

continuity by making possible to deploy and execute the control models (initially designed and tested by simulation) directly into the real target system.

RT-DEVS/CORBA [Cho03] is presented as a modeling and simulation framework, to support the development of distributed real-time systems. The framework supports model continuity for real-time software development from model design to performance evaluation and even to final real-time control. This approach is based on RT-DEVS formalism (discussed in section 2.4) and maps activities to each state. The authors do not mention details about real-time control part and the focus is on real-time simulation and a case study is presented.

Table 2.1 lists a summary of the comparison among the current DEVS-based real-time and embedded modeling approaches using different criteria. As it shows in the table, most of the current approaches are limited to the simulation of these systems, and none of them uses a formal approach for model continuity or Hardware-In-the-Loop simulation. On the other hand deadline definition and simultaneous events (collision handling) are not properly managed in the current approaches.

A) Model Continuity

Numerous researches have been done based on RT-DEVS theory to use it as an RT and embedded software design technique. However, most of the works are only limited to real-time simulation while a few of them tend to bridge the gap between simulated real-time models and the hardware using ad-hoc approaches (see e.g. [Hu01, Hua06, Cho03]) limited to specific case study systems or a just-as-needed technique. None of them proposed a generic framework for integration of the real-time models with the actual hardware counterparts. For more references in this area of research refer to [Zei93, Cho00, Sch00, Gli02, Li03, Hua04, Sag04, Gli04, Wai05, Hu07, and God07].

Lack of a formal transformation method from a simulation model to hardware control software is a common pitfall in all of the above-mentioned methods. The embedded coders and code generation tools do not quite satisfy our objectives, because the final code produced by

these software tools is prone to error and not necessarily represent the actual model accurately, thus require extra verification steps. The objective is to directly deploy the model developed to mimic the hardware controller as the final control software. Having this in mind, the model will go through extensive formal model checking and verification tests, and the software architecture is guaranteed to work as the final architecture. Hence, the code generation tools add the burden of extra verification phase to ensure that the generated software code performs the same function as the model itself. The other problem is that most available code generation tools do not support all the modeling constructs (e.g. MATLAB/Simulink Embedded Coder tool), thus the modeler is confined with the constructs that are supported by the code generator tool, limiting the modeling formalism's capabilities.

Table 2.1: Comparing RT and Embedded DEVS Modeling Approaches.

Criteria / DEVS Frame	Model Continuity	Hardware Interface	Deadline Definition	Simultaneous Events
DEVS-DOC [Sar01a]	No	No	No	*SELECT* function using Java threads
DEVS/MPC [Hua06]	No	No	No	*SELECT* function using Java threads
FDS-DEVS [Sha07]	No	No	No	Modeler defined priority based on P-DEVS
PowerDEVS [Ber10]	No	No	No	*SELECT* function
RTDEVS [Hon97]	No	No	t(i) function	Random order
RTDEVS/CORBA [Cho03]	yes	Ad-hoc interfaces	Using RTDEVS t(i) function	Random order
DEVSRT	yes	Formal generic user implemented interfaces	Formal deadline specifications	Modeler defined priority based on P-DEVS

B) Real-Time Deadline

The overall correctness of a real-time system depends on its functional correctness and timing correctness. The timing correctness is as important as its functional correctness especially in hard real-time systems. Therefore, an appropriate real-time system design methodology must reflect the timing properties of such systems. The classical DEVS theory and most of its variants do not provide a direct method for representing the deadline of outputs. The RT-DEVS formalism uses t(i) function (discussed in section 2.4) to verify the timing discrepancy between the activity *a* mapped to state *s*. However, this method does not properly represent timing limitations and deadlines to be used in a real-time system design method. RT-DEVS is originally designed to simulate real-time systems. On the other hand, the output production method is not well formulated; therefore each project uses an ad-hoc technique to interface the model with the external environment.

In the next section, the DEVSRT formalism is introduced, which manages this issue by adding a deadline function to the DEVS formalism. It also uses DEVS standard outputs as the outputs of the target real-time system to the hardware actuators. This will allow the system designer to clearly specify the deadline for each output. The system must meet the deadline requirements for each specific output, unless unexpected circumstances occur (e.g. the system overruns due to the high number of jobs inserted into the system and the system does not have adequate processing resources). I-DEVS approach (the second contribution of this thesis) tackles this problem.

C) Simultaneous Events

Authors of RT-DEVS [Hon97] emitted the *SELECT* function (of the original DEVS formalism) from their proposed methodology, asserting that even if two events are scheduled for the same time, the system will only accept them sequentially, because a physical processor can only process one event at a time. This justification is correct; however this method does not allow the modeler to control the sequence of simultaneous scheduled events in a real-time

simulation. For example, in a scenario where two or more imminent children exist (components with simultaneous scheduled internal events) within a coupled model, the coordinator (corresponding to that coupled model) does not have a pre-determined order to signal the simulators. Thus, the system executes a random or fixed order in all cases, sacrificing the reliability of the target system (which is a major issue considering the critical applications of such hard real-time systems). On the other hand, even by retaining the SELECT function of the DEVS formalism, the problem still exists based on the previous discussion in section 2.3.

DEVSRT handles this situation by taking advantage of the P-DEVS approach (refer to section 2.3) in which this conflict is dynamically handled and the modeler has the full control over the sequence of simultaneous events, or inputs.

Chapter 3: The DEVSRT Formalism

A real-time (RT) simulation is in fact an RT system that models a part of the environment or the target system and computes this model in real-time. Therefore, to use the same simulation model as eventual RT system, the simulator must be able to handle timely inputs from external environments such as hardware peripherals, software modules, network devices, human operators, etc. The idea is to develop discrete-event models using DEVS formalism, afterwards, transfer the models into embedded platform, where they function as controller interacting with the hardware through formally-defined interfaces added to the simulator. The models are thoroughly tested using various simulation-based verifications and are incrementally replaced with the hardware surrogates. This provides a Hardware-In-the-Loop Simulation (HILS) platform, where hardware and software can be designed and developed in parallel, allowing for observing un-modeled characteristics of the hardware/software designs.

Figure 3.1 illustrates the design and development cycle for control software in a plant (target platform) in the proposed DEVSRT approach. The following tasks are performed (the following numbering of the tasks correspond to the number labels on the arrows in the figure).

1. Initially the control software and the external environment (plant) are modeled together. This will allow for hardware-software co-design of the target system, where different parts of the entire system are co-modeled and tested together.

2. Once the model specifications are defined, various formal model checking approaches can be used to verify the integrity and consistency of the designed model, resulting in a robust software product[2].
3. Simulation scenarios are extracted from the target system specifications, and are used as inputs to the model.
4. The model is simulated by using the extracted simulation scenarios, investigating different aspects of the system in a risk-free environment. This process includes virtual-time and real-time simulation, where the former verifies the logical aspects of the model and the latter verifies the temporal behavior of the model.
5. The model is refined based on the results of the simulation. This provides a simulation-based verification of the entire system in which the model behavior is corrected to match the requirements of the system. This process is done concurrently with the model checking, to ensure the robustness of the design.
6. The model is partitioned into control system model and the plant model. These model partitions are then tested together in real-time.
7. The plant models are incrementally replaced with the actual hardware in the external environment, allowing for HILS of the model.
8. The control model is refined based on the results of the HILS, allowing for exploring un-modeled and hidden aspects of the external environment.
9. The incremental replacement of simulation models with hardware surrogates allows for hardware-software co-design, in which the hardware segments are initially modeled with the software components. Later the modeler decides which one goes to hardware and which one goes to software.
10. During the HILS, the model is interfaced with the hardware by using formal interfacing techniques proposed here.

[2] An initial research on integration of model checking approaches with DEVSRT is presented in [saa11]

11. Finally, the control model executive is embedded in the target system with interface functions to integrate and cooperate with the entire system.

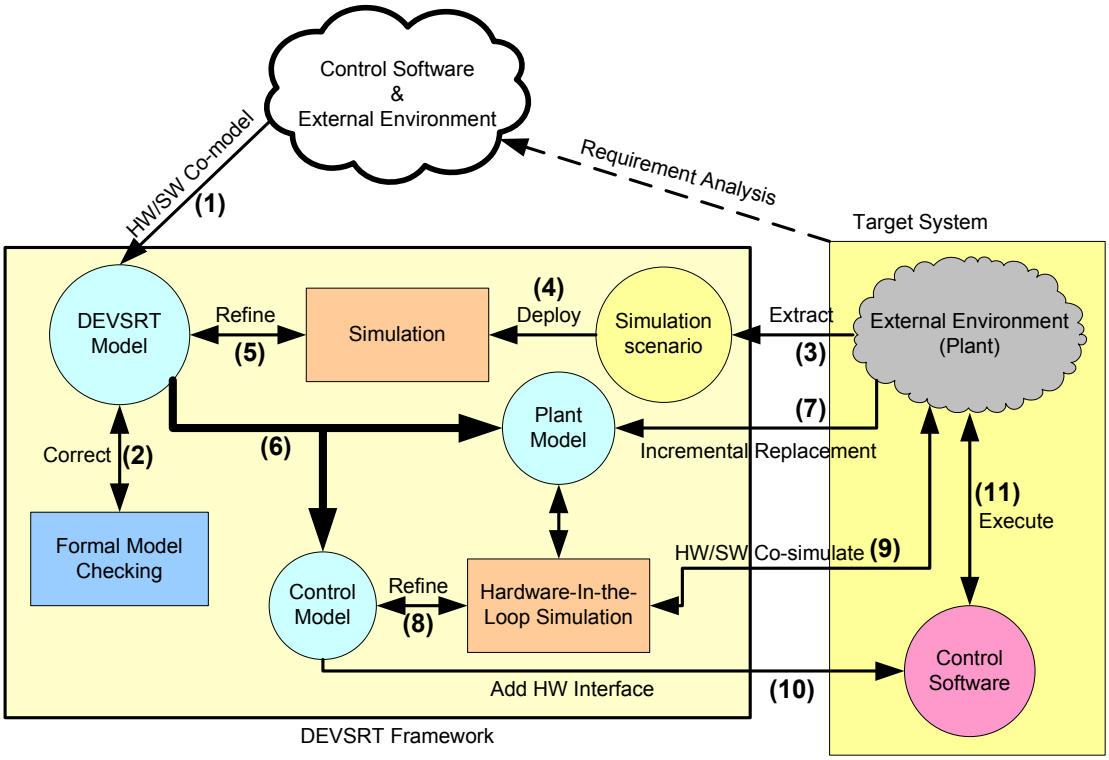

Figure 3.1: DEVSRT Development Cycle (modified from [Wai11]).

To this end, DEVSRT is proposed in this dissertation as a real-time DEVS approach built on top of P-DEVS [Cho94] methodology (to support simultaneous events in real-time) satisfying the objectives discussed earlier. Unlike RT-DEVS, this approach applies minor modifications to the DEVS formalism, allowing for easy reuse of the previous models. Finally, the proposed DEVSRT approach is implemented on E-CD++ as a tool-suit to develop real-time and embedded applications.

The most critical characteristic of a real-time system is the availability of outputs within the specified deadline. In order to express the timing constraints of the system in a formal context, DEVSRT assigns a deadline to each output in an atomic component and it verifies the deadline

when the associated output is produced. Hence, the concept of deadline is embedded in the formalism and implemented in the abstract simulation mechanism.

In DEVSRT, instead of defining an activity mapping function (as opposed to RT-DEVS) to map the state of the model with an activity on the hardware, the outputs of the atomic components are reflected to the hardware to echo the behavior of the model on the embedded device. In this approach, the output function is responsible of triggering an action on the actuator at the end of each state, informing the hardware about the new state of the model. Therefore, hardware control signals are produced by the DEVS output function.

The atomic component of DEVSRT is formally defined by:

AMRT = < X, S, Y, δ_{ext}, δ_{int}, δ_{con}, λ, ta, d >, where:

X, S, Y, δ_{ext}, δ_{int}, δ_{con} and λ are the same as P-DEVS (section 2.3).

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 deadline starts from the end of the state (release time of the output task).

To show the proof of closure under coupling of the DEVSRT formalism, it is necessary to demonstrate that a DEVSRT coupled model (CMRT = <X, Y, D, $\{M_i \mid i \in D\}$, $\{I_i\}$, $\{Z_{i,j}\}$>) can be built as a DEVSRT atomic model (AMRT = < X, Y, S, δ_{ext}, δ_{int}, δ_{con}, λ, ta, d>). DEVSRT inherited all its specifications from P-DEVS [Cho94], except the deadline (d) function. Thus, the associated atomic model derived from a coupled model will have the following specifications:

$S = \times Q_i$ where $i \in D$;

$ta(s) = minimum\{\sigma_i | i \in D\}$, where $s \in S$ and $\sigma_i = ta(s_i) - e_i$;

$d(s) = d(s_i)$;

The rest of the steps for δ_{ext}, δ_{int}, δ_{con}, and λ functions are the same as P-DEVS [Cho94].

In other words, ta(s) of the coupled model is equal to the closest $ta(s_i)$ (to the current time) of its components. Therefore, the deadline of the output of the coupled model's current state *s* is equal to the deadline of this component.

DEVSRT maintains consistency with the DEVS formalism, allowing reusing DEVS models for RT and embedded system modeling. The coupled model definition in DEVSRT is the same as P-DEVS.

3.1 Real-Time Interface

In order to make a virtual-time simulator to work in a real-time context, the logical time representation of the simulator must be tied to the underlying computing system. In a DEVS-based system, the simulation time advances only when there is an event waiting to be serviced, however in DEVSRT, the time-advance is tied to the clock of the underlying system and the Root Coordinator (RC) only verifies the timings of the events and initiates the simulation cycles based on the wall clock. In other words, RC does not advance the time, instead it waits for the physical scheduled time of the next event to reach, then triggers the event by sending the appropriate simulation message.

In order to use a DEVS model as the final target software architecture, DEVSRT employs model outputs as hardware control signals, and proposes an efficient formal interfacing mechanism between the model and the environment. A driver interfacing approach is presented in [Cho98], which is now integrated with the proposed DEVSRT in a more efficient way by removing the extra processing burden from the atomic components. In this approach, the standard DEVS I/O ports of the top-most (Top) coupled component that are supposed to interact with the environment, possess a driver object, working as an interface between the model and the external environment. This way, the model hierarchy remains unchanged and only the driver interfaces are added to the borders of the model. The driver object is an abstract function, overridden by the model developer that can be independently modified for each platform, providing portability to the model on different platforms. The other advantage of this approach is maintaining RC aware of the atomic component interactions, conforming to the DEVS abstract simulator definition by Zeigler (discussed in section 2.3A)).

The DEVSRT notation of the Top coupled model in the model hierarchy is modified as follows:

TOPCM = <X, Y, OS, IS, DX, DY, D, {M_d | d∈ D}, EIC, EOC, IC>, where:

X, Y, D, M_d, EIC, EOC and IC are the same as DEVS

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 →Xv: converts external environment input signals to input port value (Xv).

DY: Yv →OS: converts output port value to external environment output signals (Yv).

Any interaction between the environment and atomic component is routed through the formal interconnections from the Top coupled component to the atomic component or vice-versa. The interfacing mechanism allows for Hardware-In-the-Loop and Human-In-the-Loop simulation by connecting DEVS components with the hardware or human peripherals. The integration with hardware can be done incrementally, by replacing each model component with the corresponding hardware counterpart (e.g. sensor, actuator…) and providing the driver functions for the model ports previously connected to that model. The software model co-executes with the hardware segments, allowing for investigating actual environment scenarios and providing safe test-bed for each device.

Model continuity is ensured, since the original model is finally deployed on the hardware, acting as the embedded control software. The incremental hardware deployment technique provides a seamless integration mechanism, where the system is reliably embedded in the hardware. The benefit of this approach versus code generation approaches is the limitless use of the simulation model features on the hardware and no extra verification of the generated code as the same model and source code are deployed on the hardware. The only component added to the model is the driver interface functions that are gradually verified in the incremental integration steps.

The algorithm of the RC main loop in DEVSRT is as follows:

1. main():

```
2.  forever for each DEVSRT model /* main loop */
3.      wait for is signals from environment or internal time out
4.      if an external event then
5.          q = DX(IS)
6.          send (q, t) msg
7.          send (*, t) msg
8.      else if an internal time out then
9.          send (@, t) msg
10.         send (*, t) msg
11.     else if receive (y, t)
12.         OS = DY(y)
13.         send oy signal to the hardware
14.     else if receive (done, t)
15.         t_N = t
16.     end if
17. end forever
```

Lines 5 and 12 show the driver object functions that convert input and output signals, respectively. The cycle starts with the RC waiting for inputs from hardware or an internal event time out (line 3). A soon as an input is received, it is converted to a DEVS predefined input value by the input driver function. Afterwards, an external message is sent to the target atomic component (based on P-DEVS simulation mechanism discussed in 2.3A) an external message is always accompanied by an internal message). If an output message is received, the DEVS output value is converted to a DEVS signal via the DY driver interface function.

The external and internal message handling functions in the *Simulator* object are the same as P-DEVS (presented in 2.3A)). The following pseudo code represents the collect message handling function in a DESVRT Simulator object:

```
10. when receive (@, t):
11. if (t = t_N) then
12.     y = λ(s)
13.     if (t_now ≤ t_L + ta(s)+ d(s))
14.         send (y, t) to the parent coordinator
15.     else
16.         error //deadline missed
17.     end if
18.     send (done, t) to the parent coordinator
19. end
20. else if
21.     error
22. end when
```

The simulator is responsible to verify the timing of the output. Thus in line 13 the deadline function d(s) associated to the current state is called to verify whether the output is produced on time. The d(s) function returns the relative deadline of the output from the end of the current state s, thus it is added with (t_L + ta(s)) that indicates the end of the current state. If the deadline is missed an error signal is raised, informing the system about a late deadline, thus the system can decide what action to pursue.

3.2 Implementation on E-CD++

The DEVS formalism proposes a framework for model construction and also 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 it can be feasibly implemented by computer software.

E-CD++ [Yu07a and Yu07b] extends the CD++ [Wai02b] simulator (a DEVS-based framework for M&S introduced in section 2.5, and RT-CD++ [Wai04] (an extension of CD++ for real-time simulation). E-CD++ support modeling real-time systems by converting CD++ virtual time-advance function to real-time and provides an RT simulation platform for verification of such models. It also support FDS-DEVS framework [Sha07], where model components can change dynamically during the simulation. During this research, the proposed DEVSRT M&S framework is implemented on E-CD++ software, by modifying its simulation engine to execute real-time models more precisely and interacting with environment, based on the driver interfaces proposed earlier. 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. The underlying middleware is replaced with a real-time kernel and the runtime objects are imported to this platform as RT tasks. To follow the development cycle proposed in the previous section, the model development interfaces are also upgraded and several embedded functionalities are added. The rest of this section discusses the modifications carried out on E-CD++ in the context of this research to reach the DEVSRT objectives.

Figure 3.2 illustrates the E-CD++ development framework with DEVSRT modeling approach implemented during this research. This framework is a special case of the layered M&S approach presented in Figure 2.2 that is customized with DEVSRT in the modeling layer, E-CD++ as the simulator, Eclipse in the application layer, and Xenomai real-time Linux kernel as the middleware platform to execute the simulation. The target embedded platform, or HILS with the external environment is shown in this layered approach, representing the cross-platform development of models in this paradigm. The E-CD++ execution engine uses Xenomai real-time kernel[3] with multi-tasking services to implement DEVSRT. The user developed models and the driver objects are merged with the E-CD++ core objects and the entire combination is compiled to produce the model executable. Xenomai provides an RT kernel resting between the hardware

[3] Another version is also under development for embedded environments without OS support.

and Linux OS, and offers several pervasive hard RT services to user space applications and is seamlessly integrated with GNU/Linux environment.

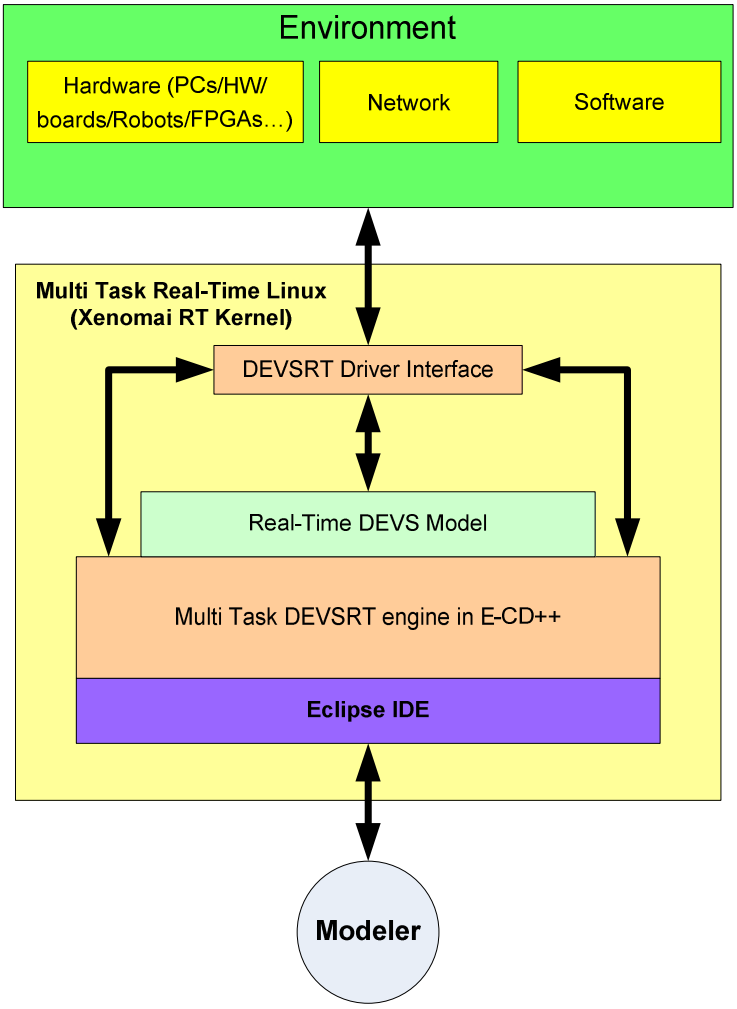

Figure 3.2: E-CD++ with DEVSRT Development Framework.

In order to improve and speedup the model development, E-CD++ incorporates Eclipse programming environment and has user-friendly interfaces suitable for real-time and embedded execution [Moa08]. The GGAD (Generic Graphical Advanced environment for DEVS modeling and simulation) [Chr04, Bon10] graphical user interface tool based on DEVSgraph standard

[Pra93] is also integrated with E-CD++ Eclipse IDE. This tool allows for graph-based drawing of the DEVS model hierarchy, interconnections, and behavior representation of atomic components to automatically generate the model-file and source files of the model. Since E-CD++ executable file is to be deployed on a different platform (embedded hardware system), means for cross-compilation for the project is also provided, as well as means of communication to the target platform in order to download executable binary files, run the executable and debug remotely [Moa08].

A) E-CD++ Software Structure

As discussed earlier in 2.5A), CD++ is modularized in which system objects run as separate software modules with well-defined behaviors and independent functionalities. E-CD++ inherited the main object entities of CD++, applying the proposed DEVSRT approach by modifying object behaviors or adding new entities to the software architecture of CD++. Four main components of E-CD++ are: Main Runtime System, Modeling Subsystem, Runtime Subsystem and Messaging Subsystem (see Figure 3.3).

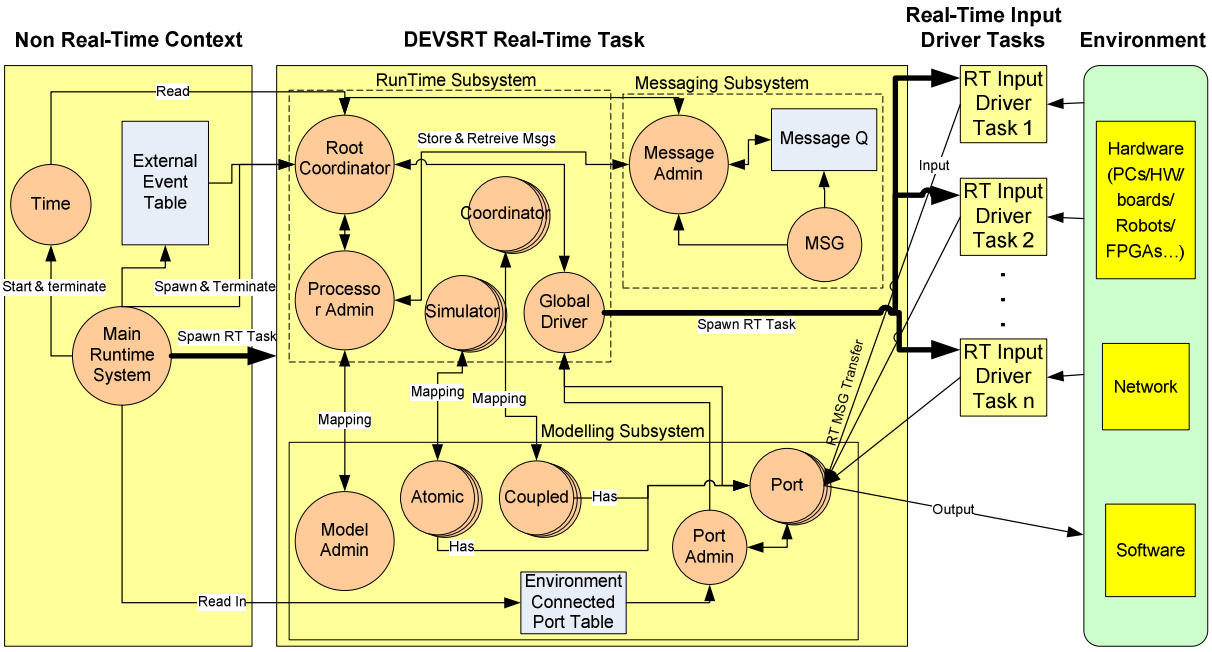

Figure 3.3: E-CD++ software structure.

The Main Runtime System manages the overall aspects of the real-time execution and provides timing functions with microsecond precision. This is done by incorporating Xenomai native skin clock functions [Xns11] in the E-CD++ *Time* class which is itself instantiated by the Main Runtime System class. The *Time* class is modified to handle microsecond time operations and also provide the physical elapsed time of the execution to be used by the RC to schedule the events. The Main Runtime System is the first object that is created in non real-time context and spawns the Runtime Subsystem as a Xenomai real-time task. In general, it does the following tasks in sequence:

- Registers Atomic component objects.
- Registers the Top coupled component ports that are connected to the external environment.
- Reads in the external events (from event-file) and builds an external event table.

- Reads in the model-file and builds the model hierarchy.
- 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 Manager. In E-CD++, The Simulators work as run-time execution engines that correspond to atomic components and perform the main job of executing transitions and output function after receiving the proper messages.

The RC is a special Coordinator that manages the real-time event scheduling. It initializes the global *Driver* object which spawns 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. Running in the context of the main real-time task, RC manages the inputs received from input driver tasks while it is waiting for the next transition time to occur and at the same time releases outputs to the output driver objects.

The Messaging Subsystem consists of the Message Manager and various Message classes. Messages are transferred to the coordinators and simulators via the Message Manager, which is responsible for delivering messages. The incoming messages are first buffered into the Message Queue and are processed by the Message Manager in real-time.

The Modeling Subsystem holds the model hierarchy information extracted from the model file. The subsystem is composed of Coupled and Atomic component classes, Input and Output port classes and the Models Manager, which maintains a hashing table of the model components and port influence lists.

The *Port Admin* object is a new entity added to the E-CD++ to maintain a list of the Top model ports that use a driver object. This list is later accessed by the global *Driver* object to spawn and control the input tasks.

The *Global Driver* object is another new entity to control the interfacing mechanism with the environment. It invokes hardware driver initialization and termination functions. After initializing the hardware, it spawns a Xenomai RT task (thread) for each input driver function.

The driver function is handed to the thread as the thread function, whose job is to receive external input signals and convert them to DEVS predefined input values. The RT thread can be periodic and the period interval is declared in the model-file. In order to synchronize the execution of the RT input tasks and minimize the jitter, a multi-value semaphore is used to signal all the tasks at the beginning of the execution, ensuring the start of the simulation cycle as close as possible to the physical start time of the simulation. It also invokes outputs port driver functions via the RC, whenever the latter receives an output message.

The initialization and termination functions of the driver class as well as the input and output port driver functions of the port class are abstract C++ functions, implemented by the user for each specific platform.

B) Performance Evaluation

In order to verify the efficiency of the implementation and to prove the performance gains of the multi-tasking approach on a real-time middleware, the proposed implementation is tested with synthetic models and compared with the previous RT-CD++ [Wai04] implementation. The tests are performed using two different sets of synthetic models with different depth and width in the model hierarchy. The results of tests are compared with the reported results of the previous evaluation of RT-CD++ published in [Gli02].

In order to compare the two implementations in an equal and fair condition, the synthetic model proposed in [Gli02] is duplicated in the new E-CD++ implementation. The model is composed of one coupled component and several atomic components in each level of the hierarchy. Figure 3.4.a illustrates the Top coupled component along with its inter-connections. The model can have multiple levels with the same architecture and several atomic components in each level. Figure 3.4.b shows the last level, which only has one atomic component.

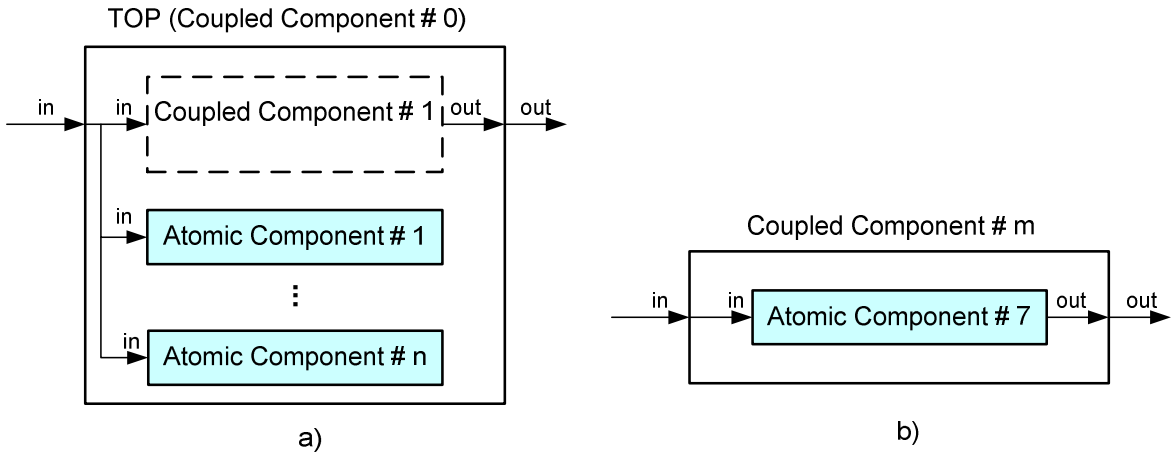

Figure 3.4: Synthetic Model Architecture (Modified from [Gli02]).

Given a specified depth d and width w, we end up having k coupled components with w-1 atomic components inside each model (except for the last coupled model, which will only include one atomic component). An input to this model propagates to each sub-component and repels to the last level. This will trigger the external function in each atomic component. All of the atomic components follow the same behavior, in which they are in a passive state, until an input is received. The external transition (invoked by the input) changes the state to a temporary state with zero time-advance, which produces an output and then transitions to a passive state, waiting for the next input. This cycle continues as long as there is an input to the system.

The goal is to measure the overhead of the processing occurred in the simulation engine proportional to the processing time of the model. The overhead of executing a model is mainly associated with the abstract simulation algorithm's message transfer scheme (refer to section 2.3A)), the handling of input and message queues and the time-advance management. The major processing in a DEVS model is performed in the external and internal transition functions in the atomic component. Hence, to produce a computation extensive model, a fixed delay of 50 milliseconds is assigned to the external and internal transition functions of all the atomic components. To make the comparison platform-independent, the models are executed on the

same workstation with the same computing power used in [Gli02]. The percentage of overhead of the system relative to the model execution time is measured and compared. The percentage of software overhead is calculated using the following equation:

$$\text{Overhead}\% = \frac{\text{TotalProcessingTime} - \text{TotalTransitionsProcessingTime}}{\text{TotalProcessingTime}} \times 100$$

3-1

Two sets of varying tests have been carried out with changes in the number of components in each level and the depth of model hierarchy (number of levels). The first test is composed of four models with fixed number of components in each level (equal to 12) and variable depths of 3, 6, 9, and 12 levels for each model. The tests were done with 10 inputs injected to the models during a fixed execution time of 40 seconds. Figure 3.5 represents the overhead percentage calculated using equation 3-1 for the above models in E-CD++ and compared with the available results of RT-CD++.

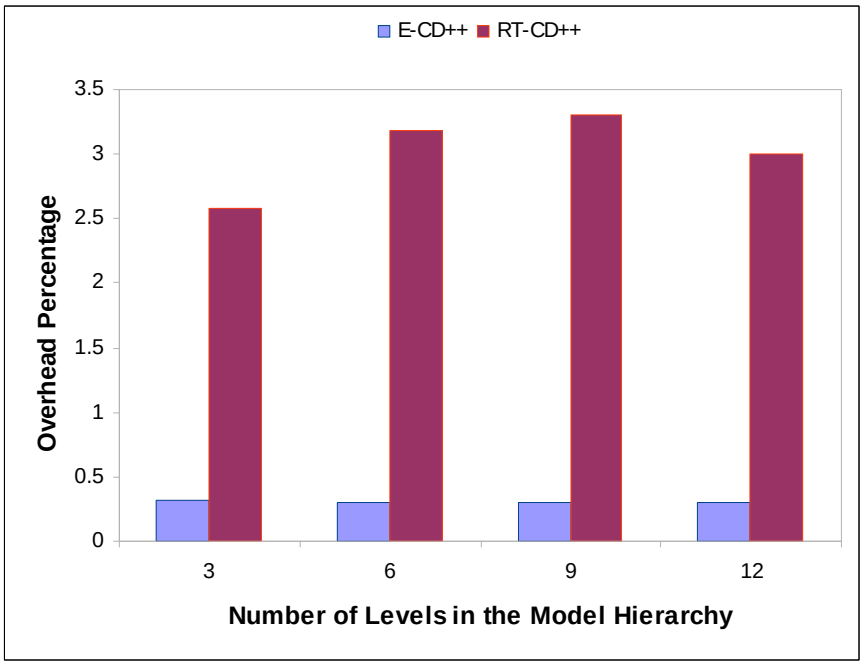

Figure 3.5: Percentage of Overhead with Variable Depth.

In another test, models of the same type are generated with fixed number of levels (i.e. 4) and variable number of components per level (i.e. 4, 6, 8, 10, and 12). The same numbers of inputs are injected to the models and the models are executed for a period of 100 milliseconds. Figure 3.6 shows the results of this test.

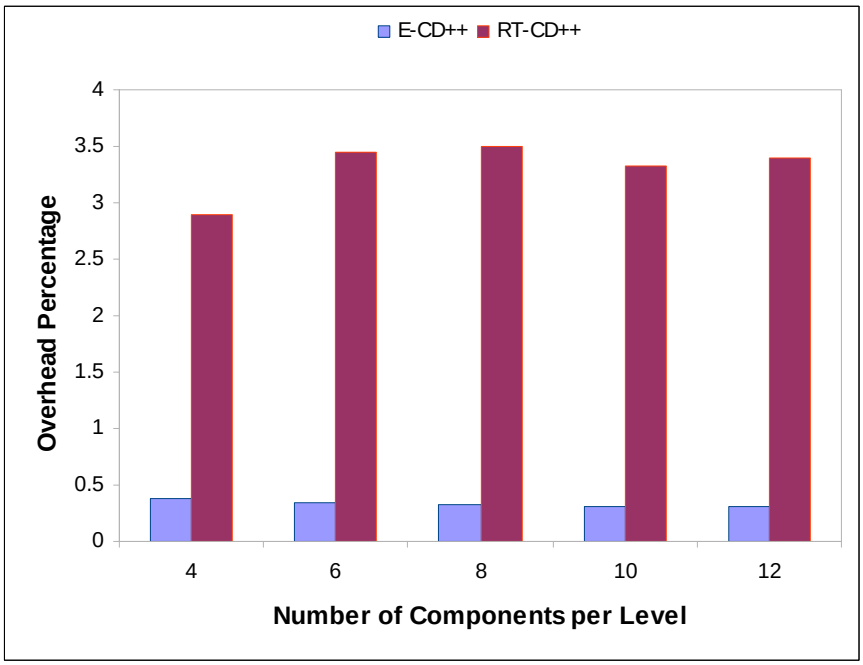

Figure 3.6: Percentage of Overhead with Variable Width.

As it can be seen from the above charts, the overhead percentage in E-CD++ is dramatically lower than the RT-CD++ in all scenarios. This is due to the use of a real-time middleware and employing a multi-tasking approach. The efficient tasks scheduling service offered by the Xenomai kernel speeds up the execution of the software, which results to a lower overhead. On the other hand, the chart demonstrates the efficiency of the simulation algorithm of DEVSRT, which does not add any significant processing burden to the E-CD++ execution engine. Another observed feature of this implementation is the constant overhead over different sizes and architectures of models. This guarantees a fixed execution engine overhead, allowing for reliable schedulability analysis of the tasks executed on the final real-time system.

3.3 Case Study: e-puck Robot Controller

As a proof of concept, a robot controller model using DEVSRT is designed and implemented on E-CD++ to perform various tests and apply different features. The model is used to demonstrate the model continuity, HILS, and hardware-software co-design contributions of the proposed methodology and tools.

The E-puck [Mon09] is a mobile robot with different sensors and motors. It has eight infrared distance proximity sensors to detect obstacles around it. There are eight LEDs mounted around the robot's body. It also has two motors connected to the two wheels on both sides, which make the robot capable of moving forward, backward, and spinning in both directions. Figure 3.7.a shows the e-puck robot and Figure 3.7.b illustrates the placement of IR sensors and LEDs on the robot.

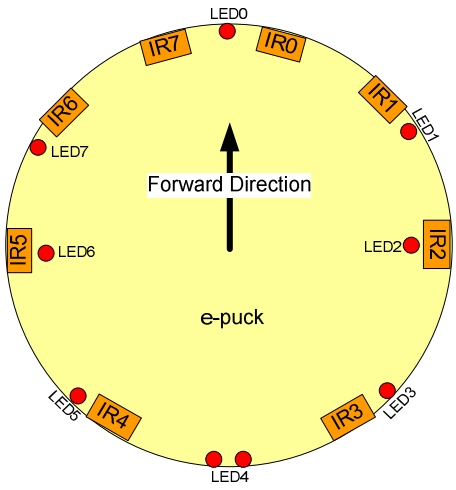

Figure 3.7: a) e-puck robot b) placement of sensors and LEDs.

A DEVSRT controller model is designed to steer the robot in a field, while avoiding obstacles. The model contains an atomic component (*epuck0*) representing the behavior of the controller and a coupled component (Top) containing the epuck0 atomic component. There are 8 input ports (*InIR0 „… InIR7*) in the DEVS model, which each of them is intended to receive

periodic inputs from a proximity sensor (the distance between the obstacle and the sensor). There are also two output ports: *OutMotor*: transferring the output commands to the motors and *OutLED*: transferring LED on/off commands to the LEDs.

The controller commands the following 5 different actions based on the inputs received from the censors: *move forward, turn 45 degrees left, turn 45 degrees right, turn 90 degrees left, turn 90 degrees right, turn 180 degrees and stop*. Initially, the robot starts moving forward while receiving the periodic inputs from proximity sensors and analyzes them. As soon as it detects an obstacle around itself, the former performs one of the turning actions based on the direction of the obstacle. The robot keeps turning until it finds an empty space on the front. The controller also uses LEDs to signal the action that is being performed. For example, if the robot is moving forward, the front LED (led0) turns on and if it is turning 45 degrees to left, led7 turns on.

Table 3.1 lists the outputs of the DEVS model and their associated actions to be performed on the robot hardware. The driver interfaces transform numeric values to the command signals on the robot.

Figure 3.8 illustrates the state diagram of the epuck0 component. The DEVSgraph state diagram summarizes the behavior of a DEVS atomic component by presenting the states, transitions, inputs, outputs and state durations graphically [Pra93]. 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 external transitions, with the input port, the input value and any condition on the input (with “port?value” format). The discrete lines represent internal transitions with the associated outputs (with “port!value” format).

Table 3.1: DEVS Output Mapping Table.

Port Name	Port Value	Hardware Command	Comment
OutLED	100	Turn all LEDs off	
	0,10,20 … 70	Turn LED off	The most significant digit is the number of Led to be turned off
	1, 11, 21 … 71	Turn LED on	The most significant digit is the number of Led to be turned on
OutMotor	0	Set horizontal & rotational speed to 0 m/s	Stop
	1	Set horizontal speed to 0.5 m/s	Move Forward
	2	Set rotational speed to 1 r/s	Turn 45°Left
	3	Set rotational speed to -1 r/s	Turn 45°Right
	4	Set rotational speed to -1 r/s	Turn 90°Right
	5	Set rotational speed to 1 r/s	Turn 90°Left
	6	Set rotational speed to 1 r/s	Turn 180°

The controller always watches for any obstacle on the front of the robot by checking the values of IR0, IR1, IR6 and IR7 sensors. Initially the robot moves forward and if there is no obstacle, it continues moving forward. As soon as an obstacle is detected by one of the IR sensors, the controller verifies IR6 sensor, and if it shows no obstacle then the left side of the robot is open hence, it performs a 45° turn towards the left side. Otherwise, it checks the IR1 value and if it is open, the robot turns 45° to the right. If both IR1 and IR6 are blocked then, it looks at IR2 sensor and if it shows an open space, the robot performs a 90° turn to the right. The same story happens when IR2 is blocked and IR5 is open, the robot turn 90° to the left. If all of the sensors are blocked, the robot tries turning to the opposite direction (180°).

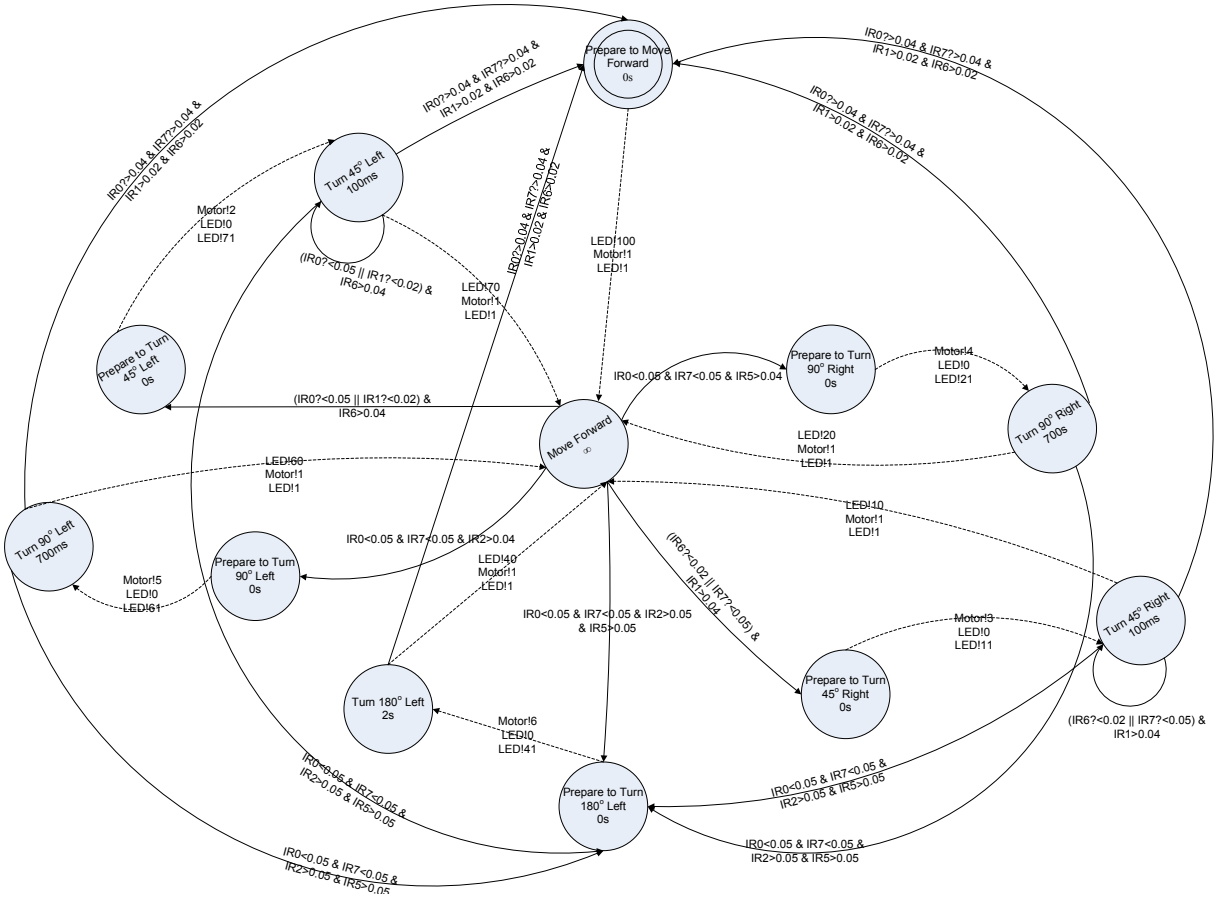

Figure 3.8: epuck0 atomic component state diagram.

As mentioned earlier, in order to program a DEVS model on E-CD++, three main components are required: 1) Model-file: where the model components, input and output ports of each component and I/O couplings are declared. The model-file is passed to the E-CD++ executable as a runtime argument and the latter instantiates model components based on the declarations in this file. It also contains information about the period of the input driver tasks and duration of states for each atomic component. 2) Source files: for each atomic component a C++ class is defined and the external and internal transitions and output function are programmed as

methods of the component class. 3) Driver interface: for each I/O port at the top level of the model hierarchy that is connected to a hardware component, a C++ port driver function is overridden from the Port super-class.

In this example, a period of 50 milliseconds is defined for the IR sensor inputs, poling the values of the IR sensors and injecting them to the model, which in response invokes the e-puck atomic component's external transition function.

Bellow is the model file of the e-puck controller model.

```
1  [top]
2  components : epuck0@epuck
3  out : outmotor outled
4  in : inir0 inir1 inir2 inir3 inir4 inir5 inir6 inir7
5  link : inir0 ir0@epuck0
6  link : inir1 ir1@epuck0
7  link : inir2 ir2@epuck0
8  link : inir3 ir3@epuck0
9  link : inir4 ir4@epuck0
10 link : inir5 ir5@epuck0
11 link : inir6 ir6@epuck0
12 link : inir7 ir7@epuck0
13 link : motor@epuck0 outmotor
14 link : led@epuck0 outled
15 inir0 : 00:00:00:100
16 inir1 : 00:00:00:100
17 inir2 : 00:00:00:100
18 inir3 : 00:00:00:100
19 inir4 : 00:00:00:100
20 inir5 : 00:00:00:100
21 inir6 : 00:00:00:100
22 inir7 : 00:00:00:100
23 [epuck0]
24 preparationTime : 00:00:00:000
25 turn45Time : 00:00:00:100
26 turn90Time : 00:00:00:700
27 turn180Time : 00:00:02:000
```

Line 1 starts the declaration of the Top coupled component and line 2 declares the DEVS components inside the Top. Lines 3 and 4 declare the output and input ports within the Top, respectively. Lines 5 to 14 define the interconnections between the ports inside the Top coupled component and lines 15 to 22 declare the period of inputs for the IR sensor driver tasks. Line 23

starts the declaration of epuck0 atomic component and lines 24 to 27 declare the duration of states within the epuck0 atomic component.

The external function performs the state transitions based on the model specifications presented above. Bellow is the source code of a part of the external transition function of the epuck0 atomic component.

```
1  if(state!=Mov_Fwd && IR0>0.04 &&  IR7>0.04 && IR1>0.02 && IR6>0.02){
2  }else if((state==Mov_Fwd)&&(IR0<0.05 || IR1< 0.02) && IR6>0.04){
3     state = Pre_Trn_45_Lft;
4     holdIn( Atomic::active, preparationTime );
5  }else if((state==Trn_45_Lft)&&(IR0<0.05 || IR1<0.02) && IR6>0.04){
6     state = Trn_45_Lft;
7     holdIn( Atomic::active, turn45Time);
8  }else if((state == Mov_Fwd)&& (IR6< 0.02 || IR7< 0.05) && IR1> 0.04){
9     state = Pre_Trn_45_Rgt;
10    holdIn( Atomic::active, preparationTime);
11 }else if((state==Trn_45_Rgt)&& (IR6< 0.02 || IR7< 0.05) && IR1> 0.04){
12    state = Trn_45_Rgt;
13    holdIn( Atomic::active, turn45Time);
14 }else if(state == Mov_Fwd && IR[0]< 0.05 && IR[7]< 0.05 && IR[2]> 0.04){
15    state = Pre_Trn_90_Rgt;
16    holdIn( Atomic::active, preparationTime);
17 }else if(state == Mov_Fwd && IR[0]< 0.05 && IR[7]< 0.05 && IR[5]> 0.04){
18    state = Pre_Trn_90_Lft;
19    holdIn( Atomic::active, preparationTime);
20 }else if(state!=Trn_180&&IR[0]<0.05&&IR[7]<0.05&&IR[2]<0.05&&IR[5]<0.05){
21    state = Pre_Trn_180;
22    holdIn( Atomic::active, preparationTime);
23 }
```

Line 1 shows the case when there is open space in the front of the robot; thereby no state change is necessary. Line 2 manages the case when IR0 or IR1 is blocked, indicating that the right side of the robot is obstructed therefore, the state of the robot is changed to *prepare to turn 45° left* and the time duration of this state is set in lines 3 and 4. The other cases and the state changes are also shown in the above code snippet. A similar program is used in the internal transition function and output function to perform the internal transitions and output functions.

As a first experiment, a random environment was modeled and the controller model behavior was observed. The model was first tested using virtual-time simulation mode, in which we added a distance generator model, which produces random IR sensor values and inputs them to the

controller model. The controller model reacts to the combination of values every one second, generated by this model. Figure 3.9 is the Atomic Animation diagram generated by E-CD++ Integrated Animation tool, which shows the sequences of input and output events in a specified period. Some of the outputs of motor and led ports are interpreted for easier understanding. The port names and scales are shown on the left panel and the values sent or received in the ports versus the time of simulation are shown on the right panel. As can be seen from the figure, at time 0 “Move Forward” output is produced by the “outmotor” port and the value of 100 is produced from the “outled” port, causing the led1 to turn on (signaling the forward moving action). After one second “ir0” receives a value of 0 indicating an obstacle on the right corner, thus the robot must turn to the left side, producing the value of 2 indicating a 45° turn to left. The rest of the diagram can be traced by following the inputs from IR sensors and the associated outputs produced in response to the inputs from “outled” and “outmotor” ports, on the timelines.

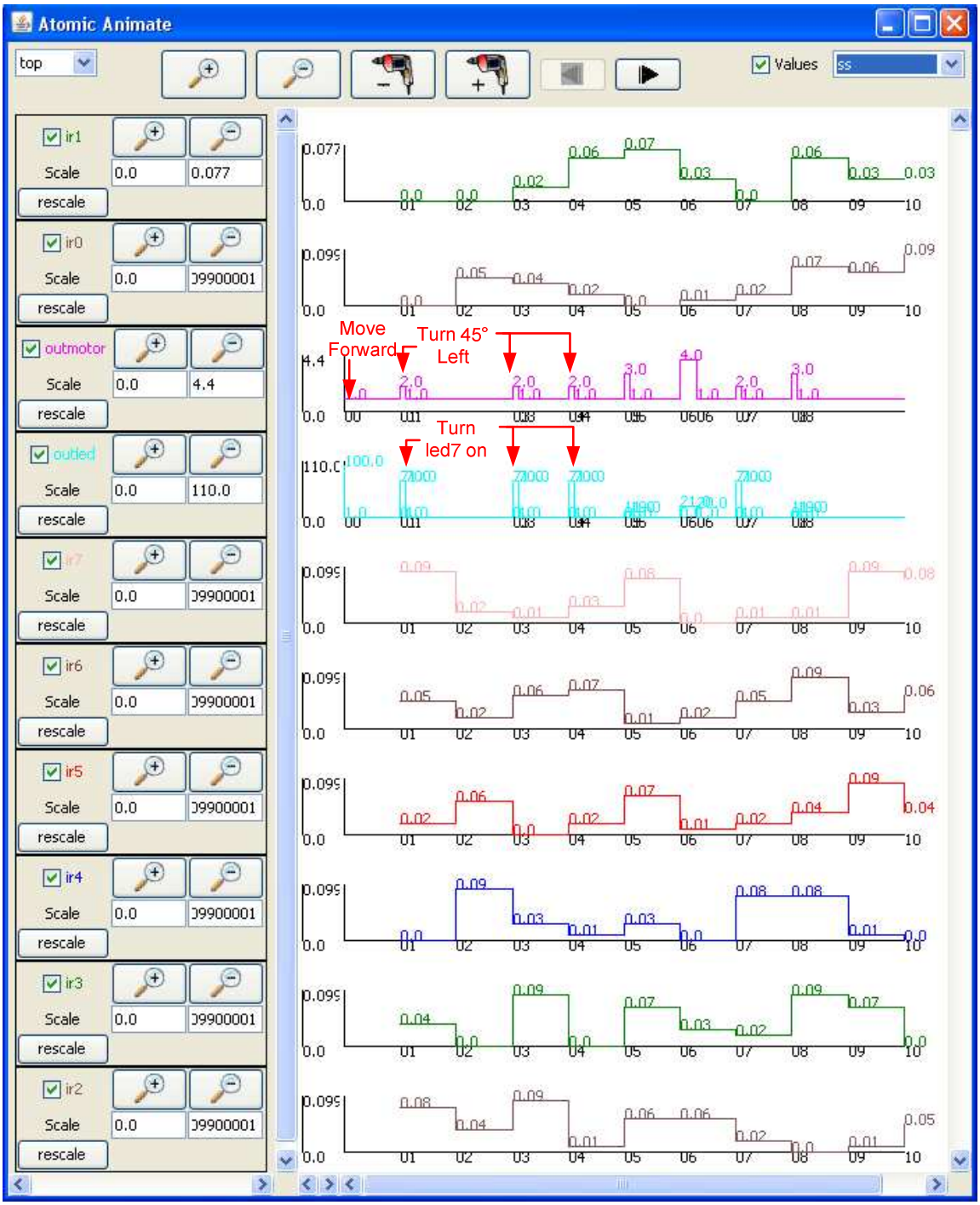

Figure 3.9: Atomic Animation diagram for e-puck random distance test.

After this test, two scenarios were designed by generating obstacles using events in the event-file. Figure 3.10 illustrates these two sample scenarios in which obstacles block the robot's path.

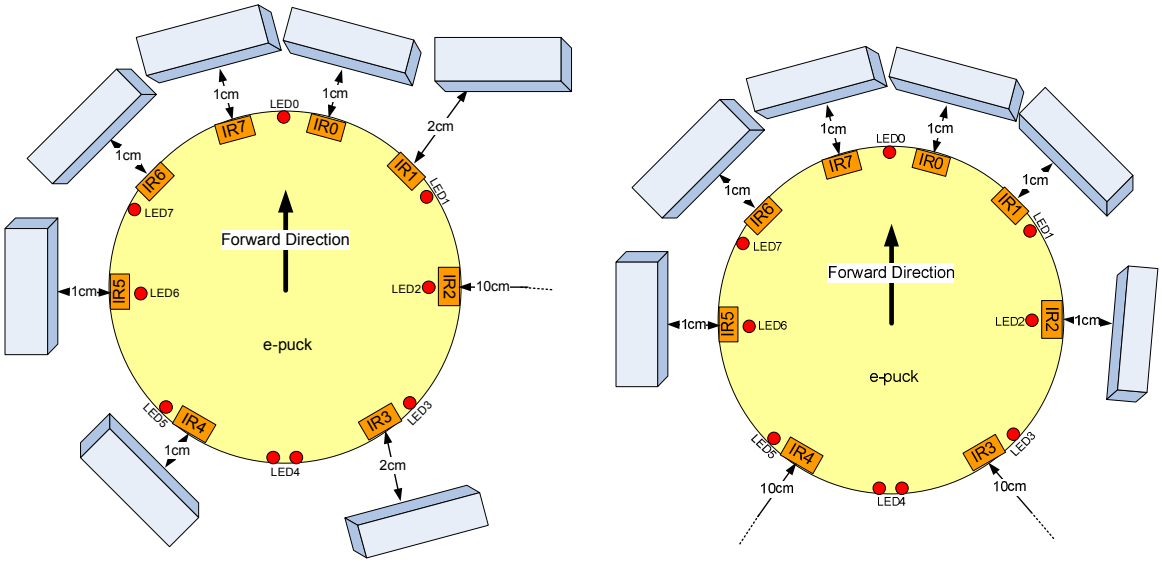

Figure 3.10: event-file scenarios a) scenario 1 b) scenario 2.

Figure 3.11 shows the I/O diagram of the model using the E-CD++ Atomic Animation plug-in. The two scenarios shown in Figure 3.10 are injected to the model at times 4 and 8 seconds, respectively. The outputs of the "outmotor" and "outled" ports are shown in the third and fourth row in Figure 3.11 and the associated commands are indicated in the figure. The robot starts with moving forward (by producing value 1 from port "outmotor"), when at time 4, the first set of inputs (Figure 3.10.a) are injected to the system (shown in the "ir0" to "ir7" rows at time 4), forcing the robot to react by turning 90° right (output value of 4 from "outmotor" port) and turning led2 on (output value of 21 from "outled" port). The second input set (Figure 3.10.b) is injected at time 8 seconds, triggering a 180° turn (indicated in Figure 3.11 in the "outmotor" row with output value of 6) and turning the rear led (led4) on (the value of 41 is produced from "outled" port).

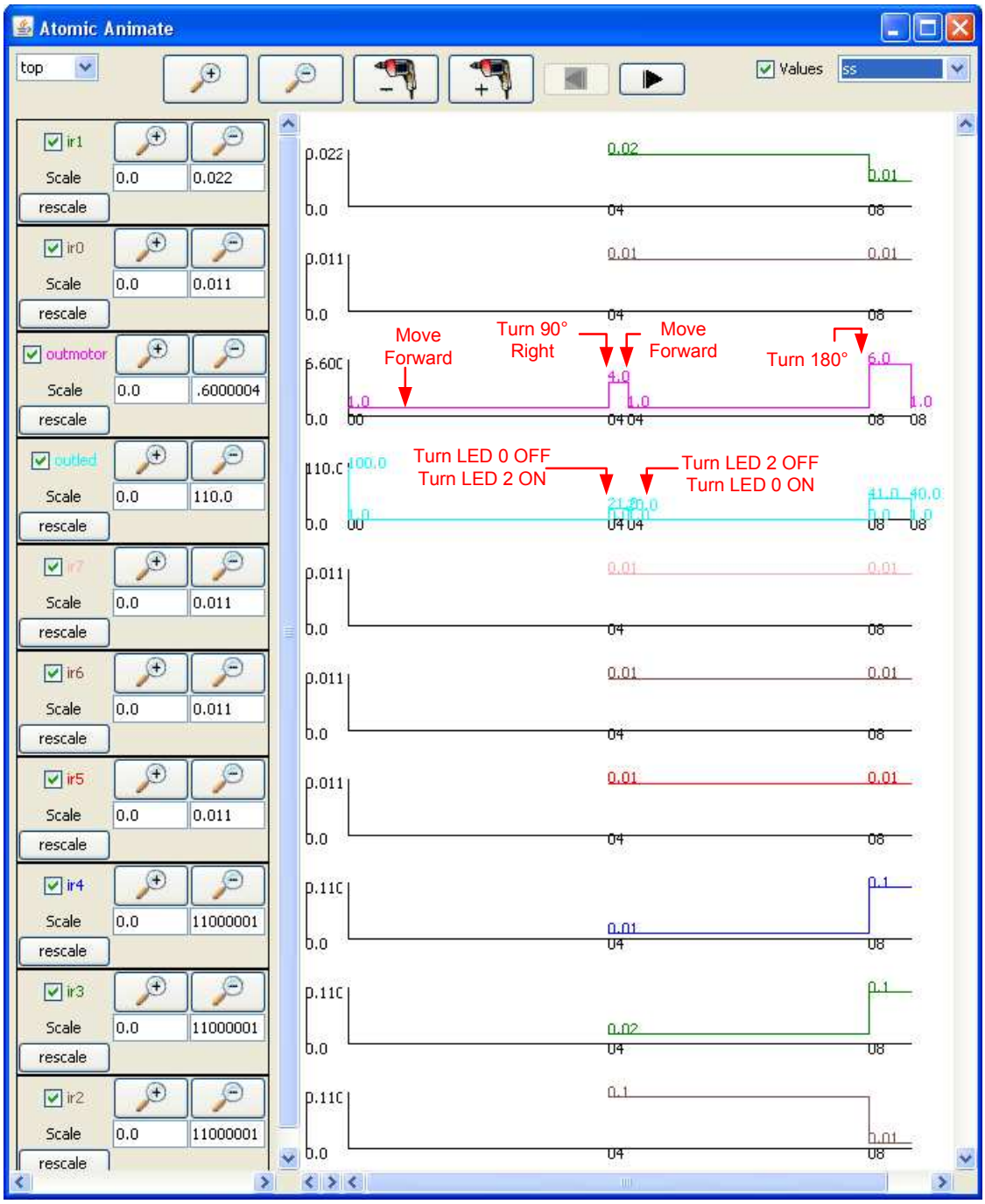

Figure 3.11: Atomic Animation diagram for e-puck random distance test.

After verifying the model behavior in various scenarios like the ones discussed above, the model was deployed in a real robot, where it was executed in real-time mode in which, the driver interfaces were activated and performed the transformation of I/O. The same behavior was observed and the robot could find its way through the obstacles[4].

This example provided a prototype of a real system designed and developed using the DEVSRT development cycle. The controller and the environment models were co-designed together; where they were tested in virtual-time simulation mode representing the hardware-software co-design approach. The controller model was also tested with virtual inputs, allowing for simulation-based verification of the controller in a risk-free environment. Finally, the tested model was deployed on the hardware proving the model continuity feature of DEVRT. Various other RT models have been designed and deployed on different hardware, such as robots, FPGA, and embedded boards (see [Moa08, Hol09, Moa09, Moa10a, Moa10b, Moa10c, and Sad10]).

[4] A video of the robot is available at http://www.youtube.com/watch?v=UFHzLk0oXyQ.

Chapter 4: Extended Applications of DEVSRT

This chapter discusses practical applications of the proposed DEVSRT approach. One of the proposed applications is developing a generic and lightweight technique comprising of an interface and a message format for deploying real-time solutions allowing for communication of DEVS models with external environments. There is a variety of DEVS-based RT simulation engines and a large number of existing models in their model repository. This technique lets modelers to reuse these models in an RT collaborative execution environment, allowing for incorporating specific services each simulation engine offers (e.g. continuous systems modeling). The formal interfacing mechanism of DEVSRT allows for integration of RT models with virtual reality engines, providing a visual representation of the simulation activities in real-time. Combining these capabilities with the embedded features of the DEVSRT, allows for visual and remote supervision of control systems developed using this framework.

The main goal is to follow a Hardware-In-The-Loop approach where RT simulators (numeric or visual simulation engines) themselves see each other as real-world devices (black boxes), interacting solely at the network messaging level. The motivations for this research are to show how to interface M&S-based systems under DEVS specifications implemented on different tools, and also a Hardware-In-the-Loop simulation with real-time visualization engines.

4.1 DEVS-Based Collaborative Modeling

A collaborative simulation consists of a source model whose components are broken into two or more groups prior to execution. These groups of components execute separately on different simulators that may or may not be implemented using the same simulation engine. The idea of

collaborative modeling using DEVS has been followed in previous works. Collaborative DEVS Modeler [Sar99, Sar01b] provides a virtual modeling workspace for expert modelers to develop hierarchical and modular collaborative models. This framework allows for model development, transformation of coupled models (federates) into simulation object, and verification of behavior synchronization among the federates. Nevertheless, this environment is limited to collaborative model development and does not implement collaborative simulation.

[Wan03] presents a web-based collaborative environment for DEVS model development. This framework uses an XML document to present the model digraph and code to different parties and allow for exchange of ideas during the modeling phase. This approach is again limited to collaborative model development, and lacks interfacing of different models together in order to build a collaborative simulation.

Another more related approach to this work is taken by [Lom06] which uses independent real-time simulation engines that accept the internal wrapping of selected components wishing to interact with other simulation tools at a high level. Wrappers hide the components and provide a means of communication with the components modeled in the external environments.

In this research, to make the interfacing visible to the modeler and allowing for dynamic exploitation of the interaction among the simulations, the wrappers are confined in the topmost coupled component. This will increase the efficiency of the system as the extra processing burden is removed from the atomic components. The other advantage of this approach is the modeler-developed interfacing mechanism, allowing for model level integration of the simulation entities. The purpose of this research is to develop collaborative models in an RT context with the DEVSRT M&S framework to provide a platform for collaborative execution of the final embedded system.

Models developed for a specific tool can be re-implemented into another tool by following their DEVS formal specification. However, this is an error prone and time-consuming approach. A more robust and scalable strategy is to keep components implemented in their original DEVS tools and make them interact over a network-centric infrastructure. In this decentralized

approach, applications resulting from the collaborative activity of the simulators (and other real-world devices) must be designed to be as robust as required in the presence of anomalies (e.g., packet loss, corruption, and sequence inversion). The participating DEVS engines do not need to worry about synchronization of time or behavior as it is handled at the model level.

In this approach, the output ports of a DEVS model are interfaced to input ports of another DEVS model and exchange DEVS formal I/O through an adapted message structure. While the simulation engines are running in real-time, different models can join this collaborative network of running models and feed from the outputs of other models while contributing their own outputs to the other models in the network. The network interface for each DEVS port can be implemented in a different way (even using different network protocols), thanks to the abstract message structure for transfer of the DEVS outputs. The component-oriented aspect of DEVS allows different coupled components (federates) in a model to operate autonomously following a common real-time clock advance. This plays the role of an implicit synchronization mechanism for event transfers between DEVS tools.

The approach delegates to the modeler the responsibility of being aware about the worst case scenarios expected for many real-world non-ideal behaviors. For instance, clock crystals of the hardware platforms hosting each simulator may drift, network latencies may vary considerably depending on load conditions, and also messages can be corrupted or delivered disordered. While these and other potential problems can be tackled by adding fault tolerance mechanisms into interfaces and/or underlying communication infrastructures, there are applications in which they can be regarded as non-critical. Hence, this approach can be categorized as a soft RT collaborative simulation approach, suitable for systems requiring a fast and best effort solution.

A) Message Structure

To implement the proposed communication scheme, a global message architecture is required. The message is supposed to carry DEVS outputs of one model to the input port(s) of other model(s) over a communication layer. A DEVS output set (Y) (defined in the DEVS

formal specification) contains the (port, value) pair, in which the port is the output port and the value is the actual output value produced by the model. We need to transfer this pair over the network and inject it to another model running on another DEVS tool as an input pair. In order to transfer this data, a lightweight message structure carrying DEVS I/O is defined with the following data fields:

Port_ID: an integer containing the destination model's input port id. Based on this field, the receiving model delivers the input to the correct input port.

Value: a character array carrying the value. The value can also be a sequence of values, sent to a specific input port in the destination model. The format of the input value is interpreted by the input interface at the destination model

The generic message structure allows for submitting different types of data between networked models. A message interface at each DEVS port of the model provides the embedding of the message as a network packet and its extraction at the destination. Each port owns an independent interface, which can be configured to submit and receive different types and formats of messages.

The modeler designs the collaborating models, having in mind the synchronization of their behavior, using a scheme to collaborate safely. Approaches and tools such as Collaborative DEVS Modeler [Sar99] can be used to design the collaborative models and verify their behavior in terms of synchronization and consistency, prior to execution.

B) Example Collaborative Model

Previous experimentation with DEVSRT models for mobile robot control applications made available a repository of target-specific low-level controllers. As controller complexity grows and new requirements arise, it is convenient to split system's design tasks into specialized and collaborative teams, reusing both experience and previously developed solutions. Following a component-based approach to demonstrate the proposed technique, the plan is to split the e-puck

robot control model (presented in 3.3) to two main components: one for the control algorithms, and the other for dealing with robot-specific drivers.

The e-puck logical controller is divided into two parts: the *Controller* and the *Driver*. The *Controller* is the main decision making unit, where the commands to avoid obstacles are generated. The *Driver* model works as a client who forwards the inputs from robot to the *Controller* and the outputs from *Controller* to the robot. The interface to the robot is part of the *Driver* model. Figure 4.1 shows an overview of the execution of the collaborative e-puck controller model running on two workstations.

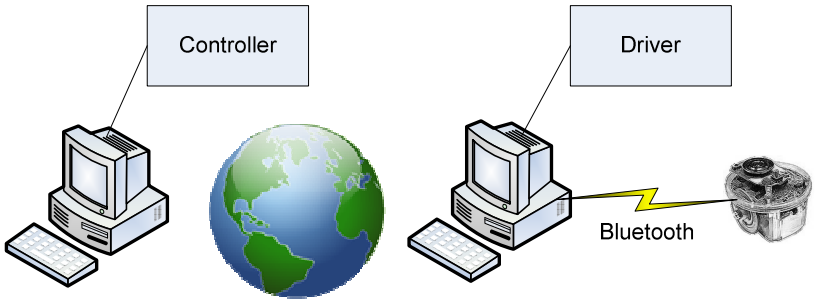

Figure 4.1: Overview of the Partitioned E-puck Model.

The e-puck robot communicates with the *Driver* model running on workstation 1 via Bluetooth connection. The *Controller* model runs on another workstation communicating through network infrastructure with the *Driver*. Figure 4.2 depicts the e-puck collaborative DEVS model details. The e-puck *Controller* receives IR sensor values from *InIR* input port via the network and sends the motor outputs to *OutMotor* output port, which is forwarded to the *Driver* model. The *Driver* receives the IR sensor values from the e-puck robot through eight input ports, and submits them to the *Controller* model by serializing them through one output port. This method reduces the network traffic while encapsulating all the values into one network packet. The *Controller* model does not deal with LED commands, and the e-puck *Driver* model generates these commands based on the motor commands received from the *Controller*.

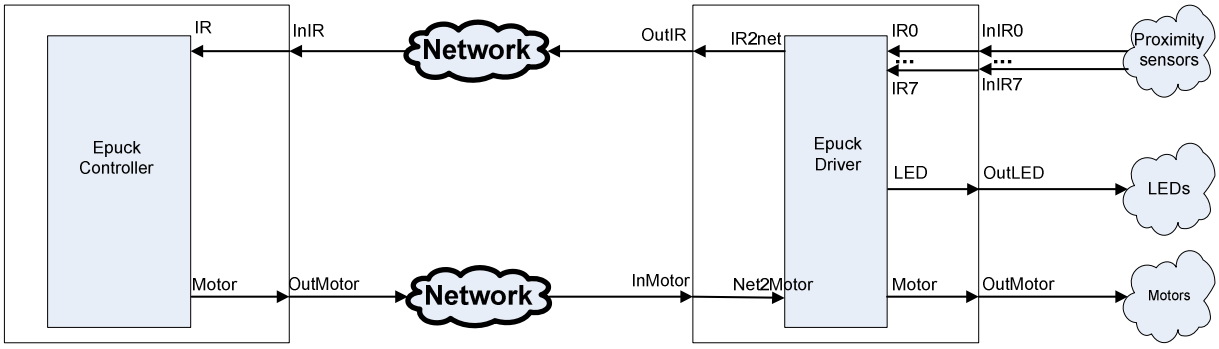

Figure 4.2: E-Puck Controller Collaborative DEVS Model.

The *Controller* model is implemented on PowerDEVS [Ber10] (by researchers from Argentina) and the *Driver* model on E-CD++. UDP network protocol is used for message transfer over an Ethernet network. UDP is chosen over TCP for its simplicity and low latency, and since the experiments were done on a local network, the chances of loosing a UDP datagram were negligible.

The DEVSRT formal interfacing technique allowed for the design of the I/O interface functions, and the E-CD++ user-implemented driver functions provided a fast development platform to implement the interfacing mechanisms. The *Driver* model's *OutIR* and *InMotor* DEVS I/O ports were interfaced with the network and *InIR0, …, InIR7*, *OutLED,* and *OutMotor* were interfaced with the robot hardware. The *Driver* is initially in *idle* state waiting for the periodic inputs from IR sensors. As soon as it receives the first value from an IR sensor, the former buffers it until it receives the inputs of all sensors. Eventually, it forwards the inputs in an array of values embedded in a network packet via the *OutIR* port to the *Controller* running on PowerDEVS. The *Driver* stays in *idle* state listening to the inputs from IR sensors and from *InMotor* port, where the motor commands are received from the *Controller*. The *Driver* generates the appropriate LED commands based on the received motor commands and forwards them to the robot. Therefore, a generic Controller model running on a different simulator with different platform is used to control a specific robot on another platform. Each DEVS output is

associated with an action on the robot (listed in Table 3.1). The driver functions of the robot output ports (*OutLED* and *OutMotor*) submit the commands to the robot via Bluetooth connection. An embedded program on the robot performs the commands on the robot hardware.

Bergero [Ber10] and Castro [Cas11] implemented the *Controller* model on PowerDEVS using an ad-hoc interface to connect the *Driver* model I/O to the PowerDEVS RT simulation engine. The two models were executed on two different workstations, transferring data through network infrastructure.

Various experiments on testing the example model were carried out. To show the results of the two simulators collaborating over a network, the log files of the experiment in E-CD++ with real-time timestamps are shown in Figure 4.3. The input log file records all the real-time incoming data (from the environment) to the model's input ports while the output file saves all the outputs of a DEVS model (with microseconds precision). The inputs and associated outputs are highlighted with red boxes in the figure. In the first box of the input file, two series of the IR sensor values inputted at time zero and after 50 milliseconds are shown (the IR sensor inputs are received every 50 milliseconds.) The first box of the output file shows the output to the *OutIR* port, which triggers the output driver associated to this port to send the array of inputs containing the values of the eight IR sensors. Therefore, when all of the IR values are received, they are forwarded to the *Controller*. Box 2 of the input file shows an input signal received from *InMotor* port containing value "1", which is interpreted in box 2 of the output file with the accompanying LED commands (added by the *Driver*). The same sequence happens in box 3 where the robot has found an obstacle and the associated IR sensor values are forwarded to the *Controller*, hence the *Controller* is instructing the robot to spin 180 degrees.

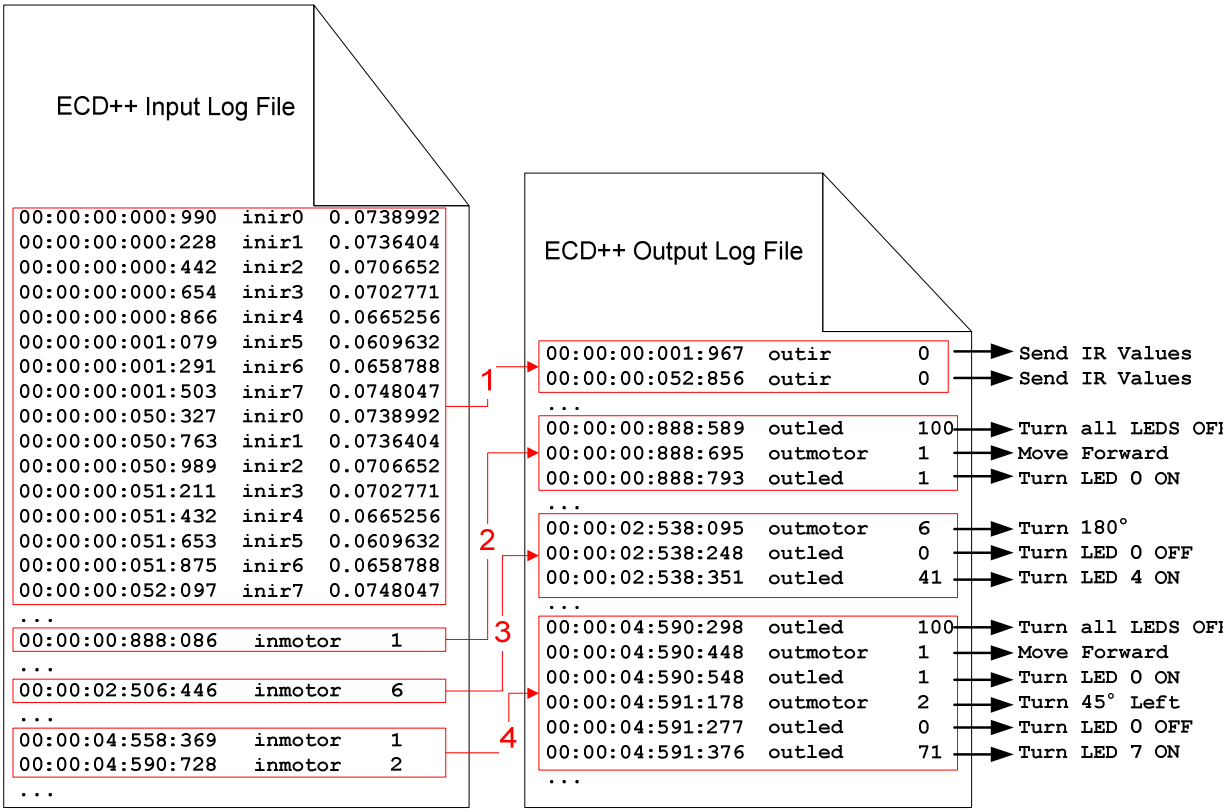

Figure 4.3: E-CD++ input and output log files.

The robot succeeded to perform obstacle detection and direction changing when the original DEVS-based system was split into two collaborative real-time models: Controller and Drivers running on PowerDEVS and E-CD++, respectively[5].

The potential advantage of interfacing E-CD++ to PowerDEVS is the collaborative execution of discrete and continuous systems under DEVS specifications. PowerDEVS provides means for

[5] A video of the collaborative e-puck model in action can be viewed online at http://www.youtube.com/arslab#p/u/12/iRqrwkPL-kQ

DEVS-based execution of continuous models, providing a great potential for RT hybrid execution of continuous and discrete models between PowerDEVS and E-CD++. This example showed a collaborative model design, implementation and execution cycle, where two development teams designed a shared model, implemented it on different tools, and executed it in a connected manner.

4.2 DEVSRT and Visualization

Integration of a computer-based virtual environment with a mathematical computer simulation provides a 3D graphical visualization, in which the user can interact with the model in real-time and conceive the model's behavior. The environment can be shared among different users and the simulation engine can also be geographically remote. This platform provides a collaborative environment for users and can be used in applications such as supervisory control, education, and entertainment. Several generic collaborative virtual environments exist (see e.g. [Car93, Pan96, Wat97, Ara08, Bou08, and Bou09]), defining their own communication scheme, which use sophisticated middleware technology such as HLA [IEE10]. The objective here is to show an example where lightweight user configurable collaboration between DEVS models and a virtual reality environment is achieved, using the interfacing delegates of DEVSRT and a specific message structure defined for this application.

In this application, an RT Cell-DEVS [Wai02a] model (implemented by Jafer [Jaf10, Moa11b]) interfaced with the e-puck robotic agent (designed in DEVSRT) and an advanced immersive visualization environment (developed by Ahmed [Ahm11, Moa11b]) for Emergency Management is developed. The emergency is handled by an autonomous robot controlled by a DEVSRT model, through interaction with the RT cellular simulation of emergencies, receiving RT data about the location of emergencies on a cell space. The immersive environment is used to visualize the emergency management activities based on the RT data received from the other two parties.

Figure 4.4 shows the system architecture of this combination. It integrates a Cell-DEVS model for navigating the emergencies, a DEVSRT emergency response model by a robotic agent, and a virtual reality component that renders the 3D scenes. The three components are designed and developed collaboratively by different researchers, incorporating DEVSRT formal interfacing technique as the principal interaction entity between the robot, the Cell-DEVS model, and the virtual reality environment.

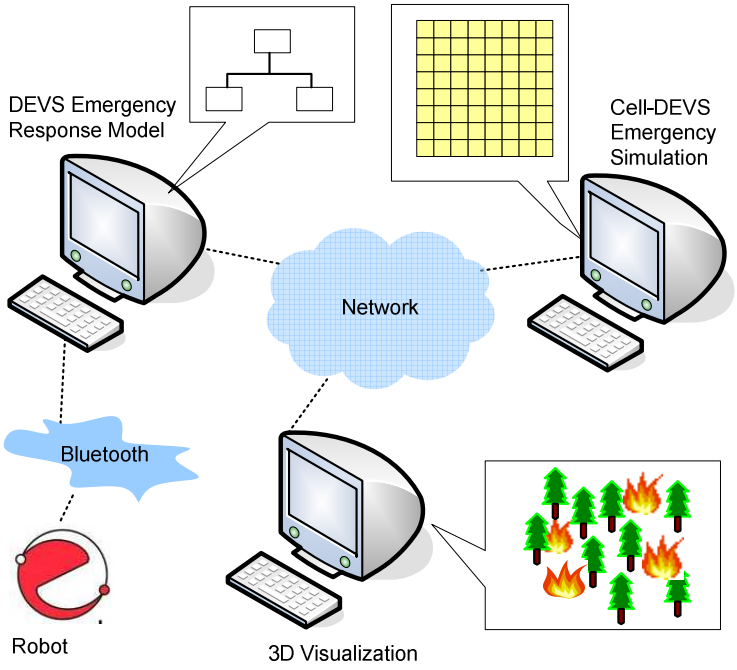

Figure 4.4: Collaborative System Architecture [Moa11b].

As can be seen in the figure, each sub-system runs on a different computer, communicating through messages transferred over a network. Figure 4.5 depicts a more detailed overview of this framework with the components in each federate and connections. The emergency simulation sub-system is in charge of the Cell-DEVS emergency model. It communicates with the DEVSRT emergency response sub-system informing it about the dimensions of the emergency area, and sends updates about the location of the emergency team on the grid. At the same time, it also

sends this information to the visualization sub-system providing it with RT data about the scene. The DEVS-based control model uses the emergency information received from the Cell-DEVS engine to respond to the emergency. Based on these commands, the robot moves on the simulated grid and deals with the emergencies one after another. The emergency response sub-system dynamically updates the cellular emergency simulation and visualization sub-system when emergencies have been resolved. This process continues until all emergencies are extinguished. The visualization sub-system produces 3D scenes of the dynamic updates received from both the cellular emergency simulation and the DEVSRT emergency response sub-system.

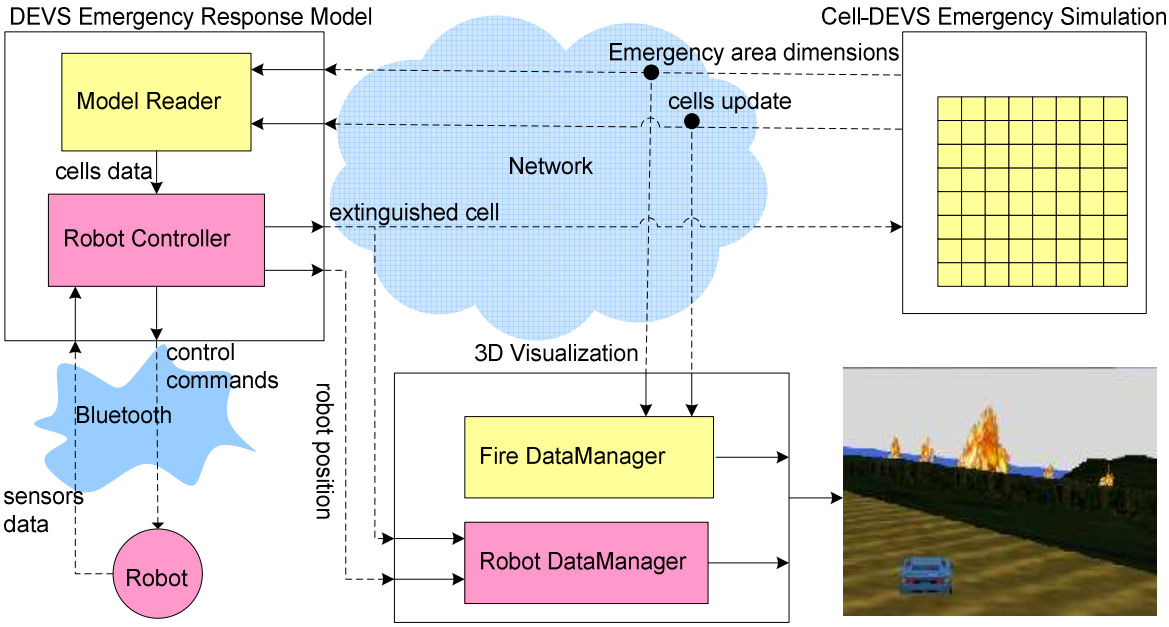

Figure 4.5: Detailed System Overview [Moa11b].

The autonomous robot interacts with the Cell-DEVS simulation engine in RT. The robot tries to reach an emergency location and extinguish it one at a time by using the cellular space as a map to navigate. Initially, the robot model receives the size of the cell space and builds a copy of the cellular space for itself. The robot also receives the updates of cell values from the cell-DEVS model and marks the changes on its own copy.

The robot controller model consists of two levels of controls: a High-level and a Low-level control, referred to as HLC and LLC, respectively. The former is responsible for path planning towards the closest emergency location (using the data provided by the cell space), while the latter is in charge of avoiding obstacles (the model specifications for this control algorithm are the same as the example model present in section 3.3. The robot model has two atomic components: the *Model Reader* and the *Controller*. The *Model Reader* is responsible for creating the local cell space, updating the cell space by receiving the updates from the Cell-DEVS engine, and signaling the *Controller* component to make path-planning decisions. The *Controller* implements the HLC and LLC algorithms, sending control commands to the robot and informing the visualization engine about the robot movements.

Figure 4.6 depicts an abstract representation of the behavior of the two components in DEVS Graph [Pra93]. The *Model Reader* starts in *wait for dimension,* where it is waiting to receive the dimensions of the cell space from the Cell-DEVS engine. As soon as this happens, it builds a local copy of the cell space, then transitions to *idle*. During the *idle* state, the *Model Reader* receives cell space updates and marks them on the local copy. If an emergency update is received, it is added to the emergency list. At the end of this state, the *Controller* is signaled to carry out the next movement.

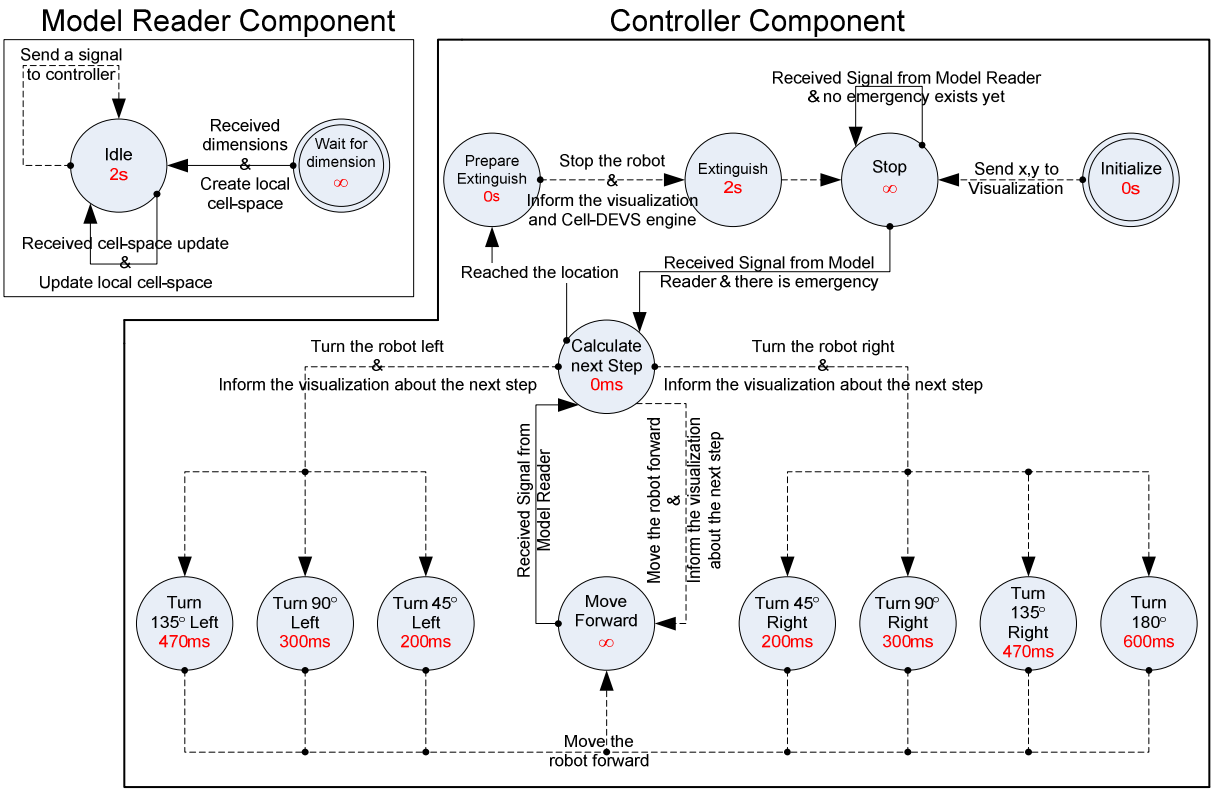

Figure 4.6: DEVS Graph of the robot controller [Moa11b].

The *Controller* starts by sending the initial position of the robot to the visualization engine and *stop*s (a state where it receives periodic signals from the *Model Reader*). If there is an emergency location in the emergency list, the *Controller* transitions to *Calculate next step* and the following tasks are executed in the corresponding external transition: sort the emergency list, find the closest site, apply the HLC and LLC algorithms, and calculate the next step. Based on the result of the control algorithms, the *Controller* changes to one of the movement states and the output function submits the movement commands to the robot, and in the next step this information is also sent to the visualization engine. This sequence is repeated until the robot reaches the emergency site. In that case, the controller transitions to *prepare extinguish*. After this, it outputs the stop command to the robot, informs the Cell-DEVS and visualization engines

about the emergency restraint, and transitions to the *stop* state, where it waits for the next location.

A) Message Structure and Implementation

The collaboration of the three components is based on a global message structure transferred over a network infrastructure. The message contains the following five data fields:

1. *msg_id*: an integer data type used to decode the type of the message and the value of the next fields in the message. There are generally five types of messages:
   - The *dimension message* carries the size of the cell space from the Cell-DEVS engine to the DEVS and visualization at the start of the execution.
   - The *robot initial location message* carries the initial coordinates of the robot from the DEVS engine to the visualization.
   - The *cell space update message* carries the cell value changes during the execution from the Cell-DEVS engine to the DEVS and visualization.
   - The *next movement message* carries the direction of the next movement at the start of each step from DEVS to the visualization engine.
   - The *extinguish message* carries the location of the emergency that has been extinguished by the robot, from the DEVS sub-system to the Cell-DEVS and visualization sub-systems.
2. *x*: used to carry the horizontal axis value (the horizontal dimension or the horizontal coordinate).
3. *y*: used to carry the vertical axis value (the vertical dimension or the vertical coordinate).
4. *dir*: carries the next direction.
5. *value*: carries the value of the cell and is used in the cell update message.

These messages are embedded in a UDP packet, and transferred during the execution of the model through the network.

The controller model is implemented on E-CD++ and the proper interface functions are programmed to send and receive the above mentioned message format. The e-puck robot [Mon09] is used as a small-scale representation of the emergency responder robot. The visualization engine is implemented using Vega Prime [Veg11] and OpenGL [Hil08] by Ahmed [Ahm11]. Vega Prime is a high-performance software environment for RT simulation and virtual reality applications. It serves as an application programmer interface (API) consisting of a graphical user interface called LynX Prime and Vega Prime libraries and header files of C++-callable functions. 3D scenes are rendered using 3D *openflight* models. The terrain model consists of trees, different buildings, roads, etc. the robotic agent is represented by a 3D truck model.

After testing the system in virtual-time, the model was executed in real-time and the behavior of the model was thoroughly verified. The robot succeeded in performing the emergency recognition based on the model discussed above and the visualization engine rendered the area and the robot movements in a real-time streaming mode[6]. Figure 4.7 shows the virtual environment in a window that is divided into two channels; one for perspective view of 3D scene (on the left), and the other channel is for orthographical view of the 3D scene which acts as 2D Map of the area (on the right). More details about this project can be found in [Moa11b].

[6] Some of the visualization videos can be watched online at youtube.com/watch?v=9aNVZRkrtC8, youtube.com/watch?v=5V4xNjdoEug, and youtube.com/watch?v=2qaWLJLVJt0.

Figure 4.7: 3D Visualization Engine Zoomed Map [Moa11b].

Another interactive virtual reality project implemented using this framework is presented in [Ahm11], where Cellular Agent model (VCELL) designed by Ahmed is used for simulating land combat[7] and is collaboratively simulated using a Cell-DEVS agent model and an advanced visual immersive simulation environment.

[7] However, this dissertation does not encourage or support military related research by any means.

Chapter 5: Imprecise DEVS

The Imprecise Computations (IC) technique [Liu94a] is a useful approach for handling real-time scheduling issues under transient overloads. It introduces a formal method of separating the critical (mandatory) part of a task from its uncritical (optional) part, thus making it possible for a real-time system to accept more tasks to the system while trying to run as many optional subtasks as possible. In this way, the system can be dynamically configured to accept more tasks when the system's processing traffic is high, while producing less accurate results, versus executing tasks completely and producing accurate result, when the system burden is low.

"A task is monotone if the quality of its intermediate result does not decrease as it executes longer" [Liu91]. Monotone tasks can be found in almost all areas of computation and their flexibility in terms of duration of computation helps designers to implement the IC technique. A solution to handle high processing peaks in an RT system is to divide the monotone tasks in the system into two versions of a computation: the primary, which executes longer and produces more accurate result and the secondary version which executes shorter but produces less accurate result. Whenever the deadline is short, the secondary version can be used to meet the deadline while having an acceptable result.

Contrary to monotone tasks are tasks with 0/1 constraints. These tasks must be executed to completion or not executed al all. Scheduling 0/1 constraint tasks is more difficult [Liu91].

The following definitions are used in scheduling algorithms for imprecise computations: Considering a set T = (T_1, T_2, …, T_n) of preemptable tasks, the following parameters are defined regarding each task T_i:

r_i is the time at which task T_i is released.

di is the deadline at which task T_i must be completed.

τ_i is the processing time required for task T_i.

w_i is the weight of the task T_i which is the relative importance.

Every task T_i is composed of two subtasks: *Mandatory* and *Optional*. The mandatory subtask M_i needs processing time m_i and the optional subtask O_i needs processing time o_i. Then $m_i + o_i = \tau_i$. Figure 5.1 illustrates these definitions.

A task T_i is referred to as executed when at least all its mandatory subtasks included in the associated jobs are executed (jobs are instances of tasks occurring during the execution). The optional subtasks of task T_i are available for execution only if the mandatory subtasks of T_i are executed properly. The scheduler can terminate an optional subtask at any time during its execution. Based on this definition, a perfect hard real-time system is a system in which all the tasks are composed of mandatory subtasks and a perfect soft real-time is a system in which all the subtasks are optional [Liu94a].

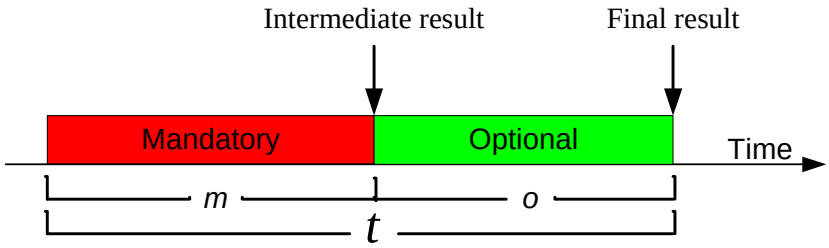

Figure 5.1: A Monotone Task Divided to Mandatory and Optional Parts [Liu94a].

IC has been well investigated and many RT scheduling algorithms have been introduced (some of these algorithms are presented in section 5.1). Imprecise scheduling algorithms benefit from the separation of mandatory and optional subtasks, using different scheduling approaches for each of them. Some of the algorithms also take into account the priority of the jobs, and there are also algorithms for periodic jobs. The concept of error is defined based on the portion of the optional subtask in each task that has not been executed.

A common problem in hard real-time systems is the occurrence of overrun situations when the system does not have enough processing resources to fulfill all the requesting processes. This issue poses critical risks to the hardware under control, and it may cause catastrophic results. In these cases, the IC technique offers an effective way of resource utilization. This chapter shows how to use this technique for the proposed DEVSRT framework by introducing the Imprecise-DEVS (I-DEVS) methodology. This approach combines the dynamic advantages of the IC technique with the rigor of a formal modeling methodology.

5.1 Algorithms for Imprecise Computation

Imprecise computation has been used for minimizing error in real-time task scheduling. The error is calculated as a function of the amount of discarded optional processing as a result of overrun situation happening in the system. Many off-line task scheduling algorithms have been proposed in the past, based on IC technique (refer to [Shi91, Liu95, and Ayd99]). There is no optimal algorithm that minimizes total error in on-line RT scheduling systems, when a feasible schedule exists, because of the lack of a-priori knowledge of the occurrence time of the jobs [Shi92].

The mandatory first algorithm assigns processing times to mandatory tasks first, based on statistics to reduce the total error (refer to [Bar98 and Chu90]).

The NORA (No-Off-line tasks and on-line tasks Ready upon Arrival) algorithm [Shi96] is based on EDF (Earliest Deadline First) algorithm and is mainly designed for online task systems, in which each task is ready upon arrival. Each task is assigned a reserved interval based on reverse scheduling algorithm. Each task's mandatory subtask is assigned a reserved execution time starting from its deadline equal to the amount of processing time required for its mandatory subtask, based on the EDF algorithm. As long as the mandatory task set is feasible a reserved interval set can be found.

The DOT_Sched algorithm [Che09] is an extension to the NORA algorithm for online tasks that are ready upon arrival. It uses three reservation lists: R(M) for mandatory tasks, R(O) for

optional tasks and R(M+O) for both of them. Like NORA algorithm each task is assigned a reserved interval in R(O) or R(M) and R(M+O) lists, starting from its deadline way back equal to its processing time.

RT-Frontier [Kob04] is a real-time operating system that presents an imprecise computation framework for constructing real-time applications. It decomposes computations to mandatory, optional and wind up parts. The wind up part works as a termination function after the termination of an optional part, reducing the termination cost and increasing portability. A novel scheduling algorithm called Slack Stealer for Optional Parts (SS-OP) is used in RT-Frontier which is based on the above mentioned three segment imprecise computation model and imposes small overhead to the system, while applying dynamic load balancing scheme.

Except the work presented in [Kob04], which only applies IC in a specific operating system, no research aims at integrating this technique with a formal methodology to be used in real-time and embedded system design and construction. The proposed I-DEVS approach, allows the model designer to deploy this technique at the design time, specifying the optional and mandatory behaviors of the target system.

IC has been applied to different fields, including RT and embedded systems [Liu94b, Kob04, and Wie08], multimedia processing [Fen93, Hua95, and Chen97], planning and artificial intelligence [Zil93, Fuj99, and Par02] and databases [Han00 and Ami03]. Despite all of these efforts, IC is not yet widely used in industrial embedded applications. The reason might be related to “strict theoretical assumptions and the lack of cost-effective support method that can be easily implemented in embedded systems” [Kob04]. This dissertation attempts to address these issues in the context of real-time DEVS-based systems.

5.2 DEVS Task Model

The main computations in a DEVSRT runtime engine occur during state transitions (modeling subsystem) and message transfers (control subsystem). Assuming the message transfer is an overhead associated to the context switching between the standard DEVS tasks in

the system, the set of tasks in a DEVSRT system (in theory) is composed of transitions and output function. This information is used to map the DEVS functions (δ_{ext}, δ_{int}, and λ) run by RT simulator into an RT tasking system, where we obtain a platform that IC can be applied. Figure 5.2 shows the processing tasks in a DEVS atomic component during a state transition. The external transition (*X*) is mapped into a task that initiates the state *S*. The task release time is equal to the arrival of the input to the model (from the external environment in the case of the topmost coupled model, or when the output generated on an output port is received in the atomic model input port). We assume no deadline for the X task. The output (λ) and internal transition tasks (*I*) are considered to execute together (task λI, as outputs in DEVS are always followed by an internal transition). The release time of task λI is equal to the end of the state ***S*** and specified by ta(s) (indicator *T*). Its deadline is specified by the function d(s) (indicator *d*) that we added to the atomic model definition.

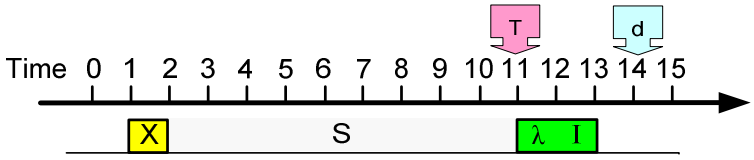

Figure 5.2: Processing Carried for a State Transition.

Now, in a scenario when a flock of inputs are injected to the system, the system has to perform a pile of transitions and produce outputs. In this case, the output tasks are drifted to later times, which might exceed their deadlines. Figure 5.3 shows an overload scenario where four inputs are injected, starting external transitions on different atomic models. As can be seen, $\lambda 1$, $\lambda 2$ and $\lambda 4$ meet their deadlines; however, $\lambda 3$ and $\lambda 2$ (second instance at time 18) do not. The internal transition *I2* produces a new state with ta(s) equal to 4 time units, which exceeds its deadline at time 17. This situation was preventable, provided that, the system could detect the overload conditions early enough to apply IC-based scheduling.

In order to execute these models, we extended the DEVS atomic model definition by dividing the above-mentioned tasks into mandatory and optional parts, incorporating the

imprecise computation concept. Assuming *X* tasks are always mandatory, *λ1* tasks can be optional. The *λ* subtask of an optional *λ1* task can be terminated under transient overloads. In other words, during overloads, the model skips optional output functions to save time and resources for the mandatory ones. For instance, an autonomous robot in a bumpy road with obstacles (flooded with obstacle reconnaissance inputs) can discard unnecessary tasks (e.g. reporting or video streaming to the base). A similar scenario can occur in any RT system where a sequence of optional outputs can be skipped to alleviate the overload situation by keeping the necessary outputs produced on-time. Schedulability analysis can be applied to this model, based on various available methods (see e.g. [Liu73]).

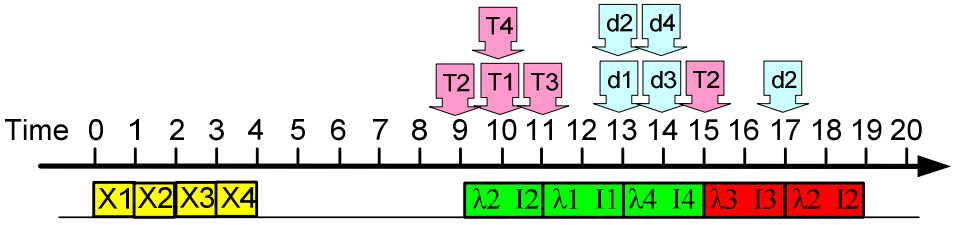

Figure 5.3: Overload Scenario.

A) Problem Statement

Based on this argument, a real-time DEVS system can be prone to overload conditions at any time during execution, proposing a significant risk and reducing the reliability of the system in hard real-time applications. The overrun situation can be very transient, happening at very random occasions when the system is flooded with RT inputs. Therefore, even by strengthening the hardware and processing resources of the system, the problem can still exist and happen at high processing peaks. The nondeterministic nature and lack of a-priory knowledge of the occurrence times of the jobs (especially sporadic jobs [Liu00]), adds to this issue severing the risk. Therefore, the need for a more robust mechanism is inevitable. It is almost impossible to design a fully trusted RT system in a perfect nondeterministic situation [Liu00], however

heuristic and mitigating techniques can be incorporated to reduce this risk and reach an accepted level of reliability.

Having said this, IC approach can be a natural choice in solving such problems in a real-time DEVS context. Defining a clear theoretical tasking system for DEVS provides a reliable starting point to employ various available techniques and algorithms for reliable scheduling and also schedualibility analysis of these systems. The proposed DEVSRT approach with minimal modification to DEVS, allows for well-organized integration of IC technique with DEVS. The rest of this chapter will elaborate the idea and present the results.

5.3 I-DEVS Formalism

Despite the theoretical advances in imprecise computation, there are few practical projects aiming at producing effective RT tools based on this technique. As mentioned earlier, the objective is to provide an imprecise framework for applications where the job arrival times are not known a-priori. The approach tries to balance the computation when the system is busy and on the other hand not reducing its performance, while keeping the run-time overhead of the implementation as low as possible.

To satisfy these goals, Imprecise DEVS (I-DEVS) is built on top of DEVSRT. The atomic model definition is modified by adding a mandatory or optional condition for each state, as follows:

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

Where $X, Y, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda$, ta and d are the same as in DEVSRT,

$S: \{(s, c) \mid s \in Z^{+}_{0}$ and $c \in \{\text{mandatory} \mid \text{optional}\}\}$.

The states of the atomic model are categorized as mandatory and optional. A mandatory state will have a mandatory output function (represented as output task) and an optional state will produce an optional output task. This abstraction in the definition of mandatory and optional tasks in the level of state machine, allows the modeler to define imprecise models without being involved in the details of the lower level tasking system.

The main runtime algorithm performed in the Root Coordinator (RC) (the top coordinator in the DEVS abstract runtime hierarchy), is unchanged. It is started first by waiting for an external or internal event. RC routes external input through an external message (q) to the destination atomic model (which triggers the δ_{ext} function). Otherwise, it waits for the closest internal event ($\lambda 1$) to send collect (@) and internal messages (*) to the target atomic model. The collect message executes the λ function on the atomic model and the internal message executes δ_{int}. The atomic model responds to the @ message by executing the λ function and returning the output value through an output (y) message. The atomic model also executes δ_{int} in response to a * message, and returns its next internal event time by a done message.

Whenever there is more than one internal event to be serviced, the mandatory ones have priority over the optional events. If an optional internal event is to be serviced later than its release time plus a grace period, its output will be discarded. The grace period depends on various factors and it defines a threshold for tolerating lateness in processing optional tasks. As discussed earlier, when the system gets busy, the tasks are drifted later from their release times. This situation can be seen as an indicator of an overload situation in the near future, triggering the system to react to the conditions. The threshold at which the system starts reacting by dropping the optional tasks is determined by the grace period. It can be a function of the processing resources, level of criticality of the optional tasks, and system's attitude towards reacting to such conditions. Grace period can be used to tune the system to obtain desirable tradeoffs between losing accuracy and meeting hard deadlines. A system with hard real-time tasks can have a shorter grace period in order to save time for mandatory tasks by sacrificing optional ones and gain more reliability, while a system with soft real-time nature can tolerate more delay in order to achieve more precision and quality. The grace period can also be modified dynamically by the system, employing intelligent learning algorithms to adapt to changing conditions. On the other hand, these dynamic conditions can be explored by deploying RT simulation using the DEVSRT HILS advantage. This way, the system can be verified and tuned in a risk free environment with various test scenarios before it is deployed in action.

The early reaction strategy helps the system to save time for later mandatory events that have not been released yet. Whenever a sequence of optional events in an atomic model is delayed, the atomic model starts discarding the output functions. The following pseudo code shows the execution algorithm in an atomic model, when receiving a collect message.

```
1.  Receive (@, t)
2.  if (s is optional AND t_L + ta(s) + t_g < t_now)
3.      raise error //optional tasks dropped
4.  else if (t_now > t_L + d(s))
5.      raise error //deadline missed
6.  else if (t_now ≤ t_L + d(s))
7.      y = λ (s)
8.      send (y, t) to the parent coordinator
9.  end if
10.     send (done, t) to the parent coordinator
11. end collect
```

Line 2 verifies if the optional output is going to be executed later than its release time (t_L + ta(s)) plus the grace period (t_g). Line 4 verifies the deadline condition and finally line 6 is the case when the output function is qualified for execution.

A) Example

Figure 5.4.a shows a simple I-DEVS model hierarchy where two atomic models *B* and *C* are coupled into *D*, which is itself coupled with atomic model *A*. Various input/output ports are used to connect these models in the figure. Figure 5.4.b shows the description of model *A* using DEVS Graph [Pra93]. Note that continuous lines indicate external transitions and dashed lines indicate internal transitions. As it can be seen, the model is initially in state *A1* (with time advance = infinity) until an input *xa* is received on port *InA.* In that case, the external transition produces a

state change to *A2*. The model stays in this state for *1t* and its deadline is *4t*. When the time is consumed, it produces the output *y2a* and transitions to *A3* (internal transition). A similar scenario can be seen in states *A3*, *A4* and *A5*, with outputs *y3a*, *y4a* and *y5a* produced respectively. Figure 5.4.c and d show the DEVS Graphs for atomic models B and C, respectively.

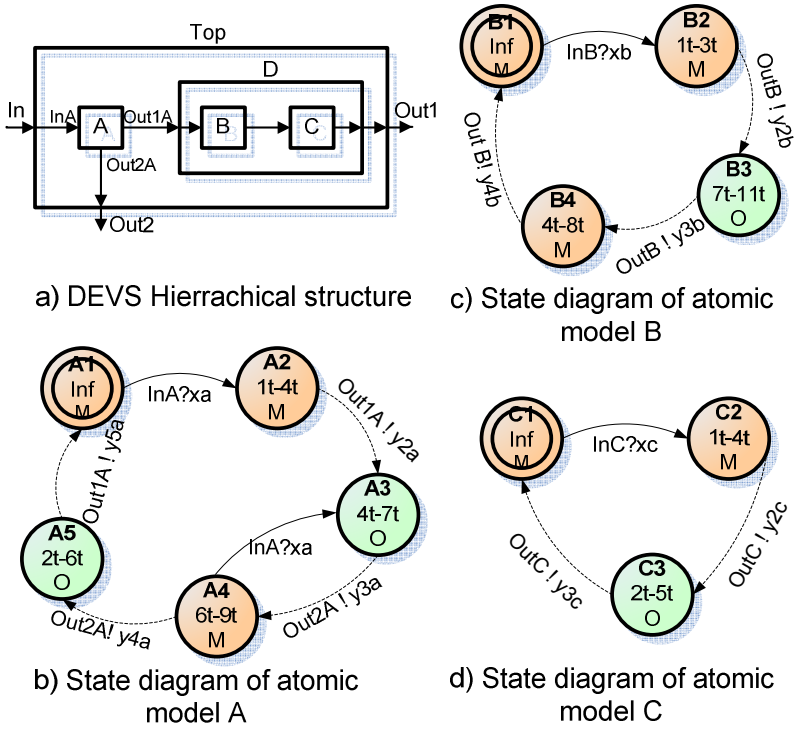

Figure 5.4: Example I-DEVS Model.

The DEVS Graph of atomic model C (shown in Figure 5.4.d) is mapped to DEVS specifications as follows:

C = < X, S, Y, δ_{ext}, δ_{int}, δ_{con}, λ, ta, d >, where:

X = {(InC, xc)},

S = {(C1,mandatory), (C2, mandatory), (C3, optional)}, and S_0 = C1,

Y = {(OutC, y2c), (OutC, y3c)},

δ_{ext} (C1, e, < InC, xc >) = C2,

δ_{int}(C2) = C3, δ_{int}(C3) = C1,

$\delta_{con} = \delta_{ext}$ has priority over δ_{int}

λ(C2) = <OutC, y2c>,

λ(C3) = <OutC, y3c>,

ta(C1) = ∞, ta(C2) = 1t, ta(C3) = 2t,

d(C1) = ∞, d(C2) = 4t, d(C3) = 5t,

The mapping of atomic models A and B are similar and straightforward.

In this example, the state durations are considered very small; however, in reality they are usually longer, compared to the execution time of the *X*, λ and *1* tasks. In a system with large number of atomic models, similar overload conditions can happen at different points of time, when multiple *X*, λ and *1* tasks from different atomic models are very close to each other, causing a drift in the execution of the tasks. For instance, Figure 5.5 shows a possible overload scenario for the DEVS model presented in Figure 5.4 without considering IC technique. An input *Xa* enters the system from input port *In* at time zero. Assuming the *X* task takes *1t*, at time 1 (i.e. *1t*) the atomic model *A* moves from the initial state *A1* to *A2*. The ta(s) of state *A2* is *1t*, thus at time 2, we run task *λ212*, producing the output *y2a* (for simplicity reasons the outputs are not shown) and the internal transition from *A2* to *A3*, (as specified in Figure 5.4.b). The output produced by the atomic model *A* (*y2a*) is translated to an input for the atomic model *B*. Thus, the task *Xb* is executed right after *λ212*, causing the atomic model *B* to change from *B1* to *B2*. The models advance according to the specifications (provided in Figure 5.4) until t=18. At this point, the tasks *λ414* of A, *λ313* of C and *λ212* (of A, B and C, shown in red) miss their deadlines because of the overload condition in the system.

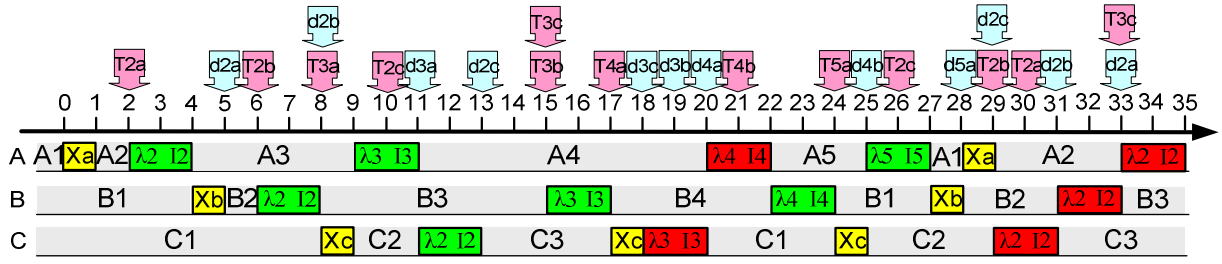

Figure 5.5: Example Transient Overload Scenario.

On Figure 5.4.b, c and d, the mandatory and optional states are marked with an M or an O, respectively. By applying the proposed imprecise DEVS technique and considering a zero grace period (a hard real-time system), *λ3* of A is skipped (because state A3 is optional and *λ3I3* is executed after its release time plus zero grace period, *T3a*), causing *λ4I4* to be shifted to time 16 and saved from lateness. The same condition happens for *λ3* of B and C. Hence, by discarding three optional *λ* tasks the four mandatory tasks and their associated outputs are saved from lateness.

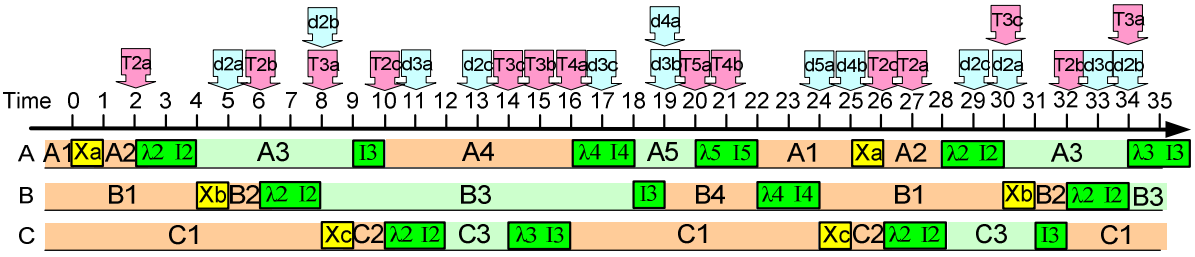

Figure 5.6: Applying Imprecise Computation to the Sample Scenario.

5.4 Results and Discussions

The proposed imprecise DEVS formalism was implemented on E-CD++ on the Xenomai RT framework. *X* tasks are made user configurable (i.e. *periodic* or *aperiodic*), and their main job is

to run user-defined input driver programs as soon as they are spawned. The main RT task implements the DEVSRT run-time abstract algorithm (discussed in 3.2) and takes care of λI tasks. This task is also responsible to implement and verify the imprecise DEVS formalism and its execution. The implementation of the imprecise computation on E-CD++ is seamless and backward compatible (i.e. the previous models also can be executed and are considered precise models).

The proposed implementation of imprecise DEVS on E-CD++ has been tested with variety of modeling scenarios and several criteria has been applied for verification of the final implementation. For instance, a synthetic robotic model with 20 atomic models, each of them connected to an external input port, connected to a sonar distance sensor and an output port connected to an electrical motor is used. To ensure that the same scenario runs every time, the values coming from the sensors were the same in all tests. All the atomic models follow the DEVS Graph diagram in Figure 5.7. The model is a synthetic representation of a robot controller, which receives inputs from sensors and based on the inputs, instructs the motors. 20 atomic models are used to make it a computation intensive model where overrun situation happens frequently. The DEVS Graph diagram in Figure 5.7 is composed of three optional states and three mandatory states. Whenever there is an input in states C, D, E, and F, the model transitions to state B. This model is used to perform comprehensive performance tests, and compare the results of the imprecise execution and precise execution. In the case of precise execution, all the states are assumed mandatory.

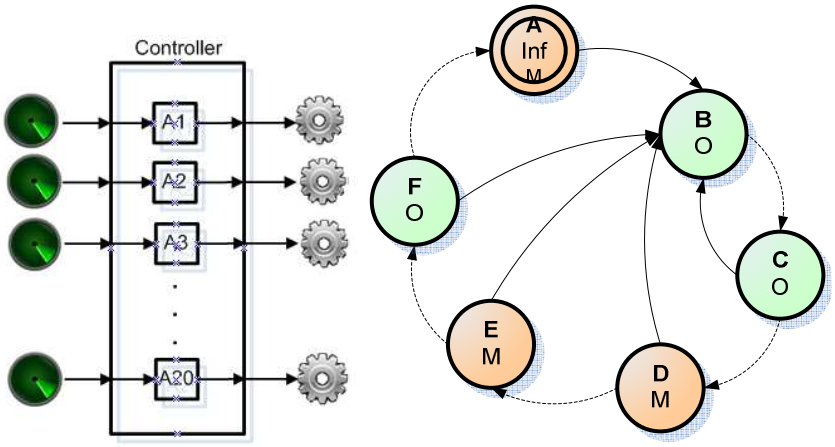

Figure 5.7: Synthetic Robotic Model Used for Verification.

The timing for the component models varied for the different tests, performed. The first test discussed in this section compared the number of discarded λ tasks versus processor utilization. The diagram in Figure 5.8 shows the results of this test, for a total execution time of 20 seconds.

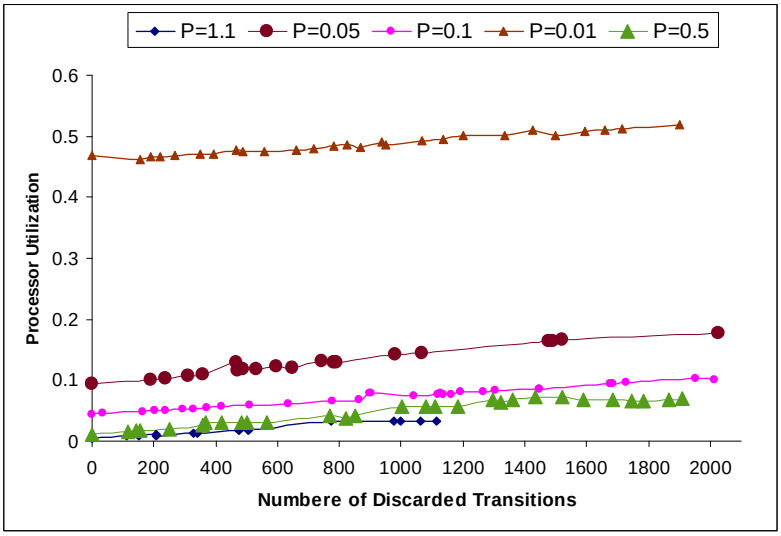

Figure 5.8: Discarded Tasks vs. Processor Utilization.

The test was performed for input period intervals of 1.1, 0.5, 0.1 and 0.001 s. As it is observed from the chart, by increasing the number of discarded tasks (which happens by

tightening the state durations and period of the inputs) the processor utilization increases linearly. The result demonstrates the integrity and persistency of the implementation in a medium load scenario. In addition, as the system gets busier the number of discarded tasks also increases. The slope of the diagram for different period configurations stays the same, showing the integrity of the functionality of the algorithm for different levels of load on the processor.

Figure 5.9 shows the average response time of all the mandatory λI jobs versus the execution time for the same model using imprecise and precise modes. In this case, the input period of all X jobs was fixed (1100 ms). The test was performed five times for each instance and the average result has been considered. As the chart shows, the average response time of the mandatory λ jobs drops dramatically in imprecise mode. In this example, there is a heavy load that the system must respond to, which required longer time for mandatory λ jobs to complete in precise mode. Imprecise computation discards the optional tasks, thus the response time of the mandatory tasks shortens.

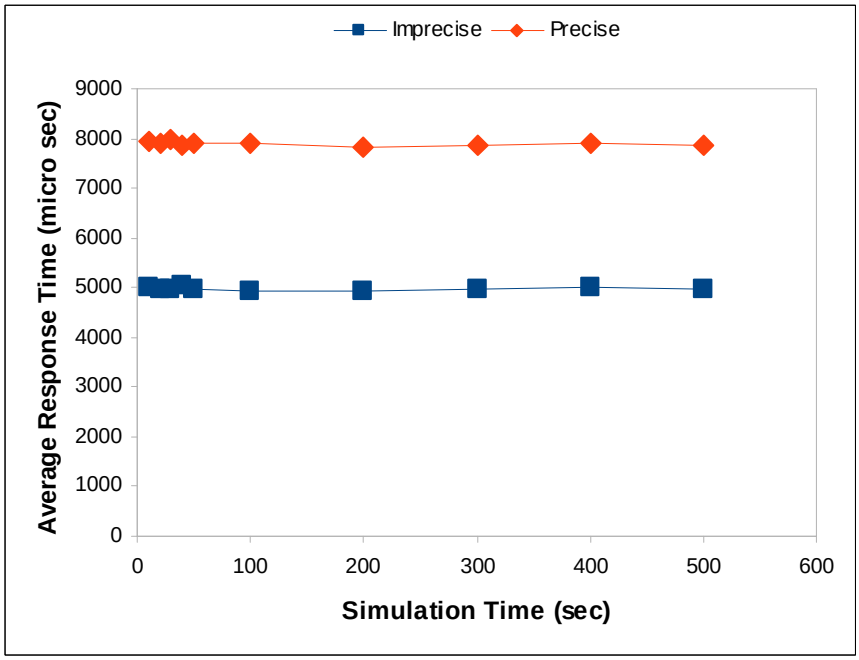

Figure 5.9: Response Time vs. Execution Time in Heavy Load.

Figure 5.10 shows the average response time of the model versus the number of discarded tasks for 20 seconds of execution time. The period of inputs is set to 50 milliseconds, and by varying the state durations, we obtain different number of discarded tasks in imprecise mode. For each instance of the imprecise test, the same configuration was applied to obtain the corresponding results in precise mode. It can be seen that the average response time of the corresponding precise execution for each instance is slightly higher than the imprecise one in medium load scenario. This is due to time being saved by discarding optional tasks in favor of mandatory ones. However, this difference is not very visible in a medium load scenario because the system is not as busy as a heavy load scenario. The chart shows that by increasing the number of discarded tasks (i.e. tighter state durations) the average response time also increases. However, this increase is not smooth as the situations change, while the system saves the mandatory jobs in transient high processing occasions, the effect of these situations on the average response time in medium load scenario is not as visible as heavy load scenario.

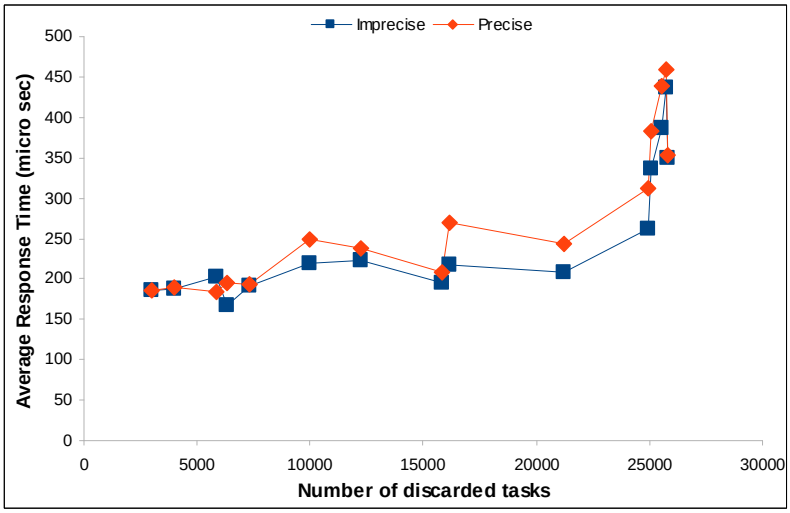

Figure 5.10: Number of Discarded Tasks vs. Average Response Time in Medium Load.

Figure 5.11 depicts the processor utilization versus the number of discarded tasks in a heavy load scenario with the input period of 2 milliseconds and 20 seconds execution time. The

chart shows steady but higher processor utilization for precise execution. The processor utilization for precise execution in all instances of the test is almost full, therefore as the load increases; the utilization remains almost the same. However, the imprecise processor utilization is instable and decreases as the number of discarded tasks increases. This is due to the instable and varying conditions that occur in a very heavy load scenario in imprecise mode. As the number of the discarded tasks increases, less processor usage is required. This decrease is not smooth neither linear, because of the change in conditions in each run, and admission of more mandatory jobs. Nevertheless, the system is successful in opening space for mandatory jobs by discarding optional ones.

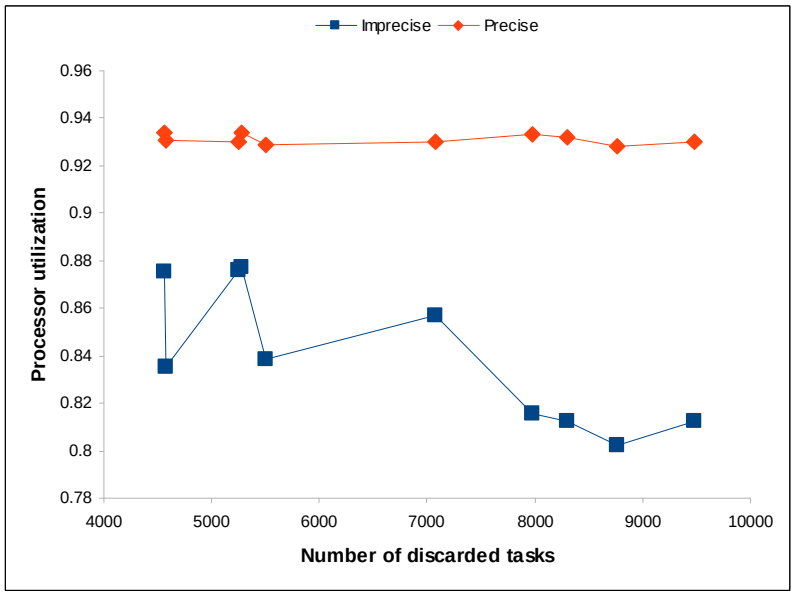

Figure 5.11: Number of Discarded Tasks vs. Processor Utilization in Heavy Load.

In a different test scenario, the synthetic models introduced in section 3.2B) were used to measure and compare the performance and overhead of the imprecise and precise executions of the same models. In this test, the models with fixed number of levels (4 layers) and variable number of components in each level (i.e. 4, 6, 8, 10, and 12) were used to measure the execution overhead and response time of the output tasks. The processing time of the X, λ, and I tasks was

set to 10 milliseconds for each atomic component. Figure 5.12 shows the average response time of the tasks during an execution time of 40 seconds for different number of components per level. The output produced in each cycle of input, is marked as optional, therefore whenever the system faces an overrun condition, the λ tasks are skipped. The chart shows that the average response time of the tasks is slightly shorter in imprecise execution in each scenario, due to the time saved because of dropping the optional tasks. On the other hand, it is observed that, when the size of the model increases (the number of call to the tasks also increases), the difference between the average response time of imprecise and precise runs also increases. This is due to heavier workload produced in bigger models and propagation of data in the model, which is efficiently handled by the imprecise scheduling algorithm, reducing the response time of the tasks.

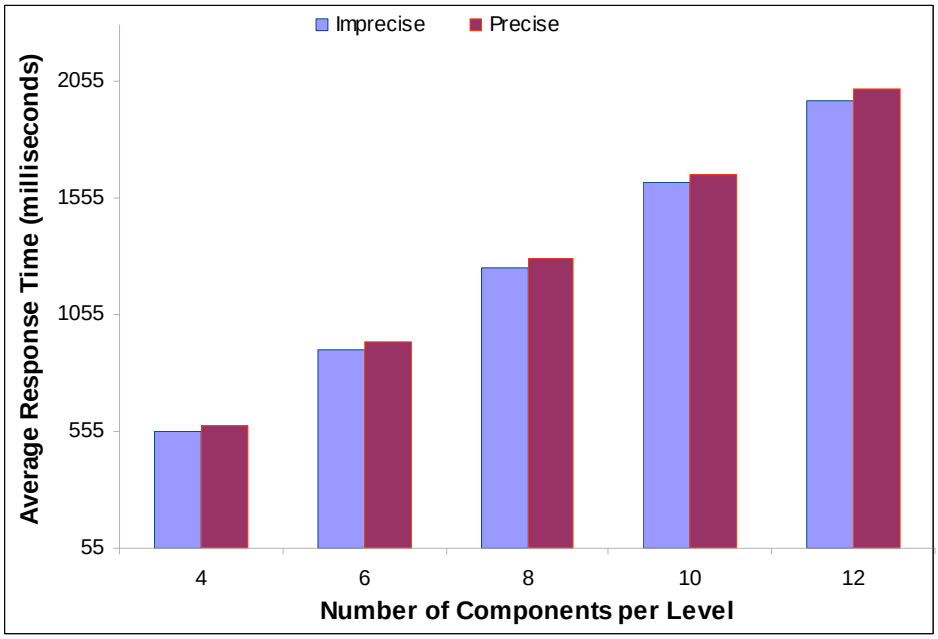

Figure 5.12: Number of Components per Level vs. Average Response Time.

The other interesting fact, observed with this test is the lower overhead in the imprecise computation in heavy processing models. Figure 5.13 represents the overhead percentage

(calculated using equation 3-1) of the execution engine relative to the tasks processing times, in imprecise and precise scenarios. The other parameters of the execution were the same as the previous test. Based on the results presented in this figure, the overhead percentage in imprecise mode is less than the one in precise mode, due to the drop of messaging overhead produced by the optional output tasks. A discarded output task eliminates the time required for transfer of (@, t) and (done, t) messages from the atomic component to the Top model. The other interesting fact extracted from this diagram is the lower overhead percentage for heavier models (bigger modeling hierarchy with computation intensive tasks), due to the increase of the task processing time portion over the execution processing time. In other words, in a computation intensive model, as the size of the model grows the overhead percentage decreases, because the processor is mainly busy with the tasks rather than the execution overhead.

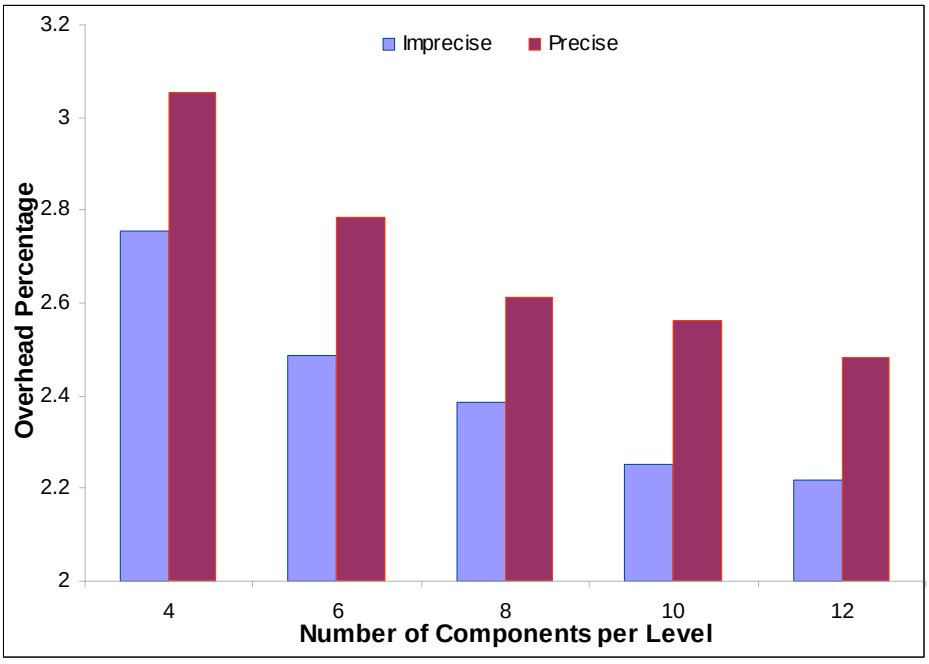

Figure 5.13: Number of Components per Level vs. Overhead Percentage.

A) Performance Evaluation

The criticality of RT systems requires efficient development approaches, in which the theoretical design and its associated implementation produce efficient throughput from the system and the resources. Various performance evaluation approaches for RT systems exist in the literature. The Rhealstone benchmark [Kar89, Kar90] evaluates the performance of an RT operating system using the following metrics: average task switch time, average pre-emption time, average interrupt latency, semaphore shuffle time, deadlock break time, and inter-task message latency. These metrics can be incorporated into DEVSRT and I-DEVS formalisms to evaluate the RT application. The M&S capability of these formalisms allows early performance evaluation of the system in simulation mode, where the system can be tested against the above-mentioned criteria. This will provide early information regarding the required hardware resources to implement the final system. The Rhealstone benchmark metrics can be applied to DEVS task system proposed in 5.2. The average task switch time between the X, λ, and I tasks can be measured in order to have a concrete idea about system overhead. The average interrupt latency can be applied to X tasks, when the system receives an input until the input is accepted by the main runtime system. The inter-task message latency can be used to approximate the delay in RT message transfer between the RT tasks in the multi-tasking implementation approach. Semaphore shuffle time can be used to measure the delay in switching the semaphore used for synchronizing the input tasks and for semaphores used by the modeller in the RT input tasks in order to avoid input resource deadlock.

The Hartstone benchmark [Ada90] developed at Carnegie Mellon University evaluates hard real-time systems using a set of operational requirements for synthetic applications. The synthetic application is used to verify the deadline requirement, whether the output meets the deadline or not. As a future work, these performance evaluation techniques can be applied to the DEVSRT and I-DEVS RT platform to measure the availability of outputs in different models using different scenarios. Scenarios can include: periodic, a-periodic and sporadic tasks, harmonic and non-harmonic state durations.

On the other hand, metrics like performance profiling, A-B timing, and response to external events can be also incorporated in model execution.

B) Scalability

RT systems working in the context of embedded hardware are prone to several limitations. One major constraint in these systems is the power consumption or battery life. High performance requirement in these systems conflicts with the low power objective. To achieve these goals performance degradation strategies can be incorporated. I-DEVS can be a natural choice for this purpose, providing a dynamic and early reaction scheme to tackle this problem. The graceful degradation strategy based on Imprecise Computations theory allows for degrading the system performance when needed by dropping the optional transitions. This threshold can include battery life or any other constraint conflicting with performance of the system.

On the other hand, performance of the system also depends on the underlying hardware. Figure 5.12 can be viewed as a scalability indicator in the current implementation of the I-DEVS approach on E-CD++ software. As the number of components per level increases the average response time increases too. This means that the tasks are executed later to their release time, when the system scales up. Likewise, this delay also affects the deadline of the tasks, thus proposing a risk. A simple solution might include upgrading the underlying hardware resources in order to solve the scalability problem. As this will be a natural solution to this problem, however the “Speed-Performance Tradeoff Anomalies” [but06] dilemma shows that in an RT system with timing and resource constraints, increasing the processor speed does not necessarily lead to a better performance, and vice versa.

Chapter 6: Conclusions and Future Work

This dissertation addressed some of the issues in the area of RT and embedded system design and development by employing an M&S-driven engineering approach. The issue of model continuity providing reuse of simulation models for the final hardware embedding has been discussed and the **Discrete EVent System Specification in Real-Time (DEVSRT)** approach is presented as a DEVS-based solution. The proposed DEVSRT was employed in developing a **lightweight collaborative RT model execution** and **virtual reality integration** framework for specific applications. Finally, this platform was integrated with Imprecise Computation (IC) technique to propose the novel **Imprecise DEVS (I-DEVS),** a DEVS-based RT task scheduling and resource management technique.

DEVSRT as an RT domain extension of the DEVS formalism provided a model-driven approach towards RT and embedded application development. The formal and intrinsic advantages of DEVS are combined with RT features to propose a design scheme for such applications. Issues such as Hardware-In-the-Loop Simulation (HILS) or Human-In-the-Loop Simulation are addressed in this framework by introducing formal interfacing mechanisms between the DEVS model and the target embedded environment. The benefits of simulation-based verification are employed by DEVSRT, allowing for pervasive verification of the system under development in a risk-free setting, exploring varying test scenarios. The concept of model continuity and reuse of simulation models in the development of final embedded software architecture are addressed. This is a shortcoming of most of the available approaches that has been solved in this dissertation. DEVSRT provides a high level abstract hardware-software modeling scheme, where different components of the target system can be modeled together. The

co-modeling approach allows for co-simulation and verification of hardware and software segments of an embedded system in a unified framework, while an incremental replacement of the models with hardware surrogates explores the un-modeled aspects of the devices with the controller component.

The formal interfacing techniques in DEVSRT enabled the collaborative execution of RT models independent from the underlying simulator platform. It also allows for interfacing DEVS models with visualization engines, collaborative control of an embedded system by co-executing different models on different RT engines and at the same time, interacting with the target hardware platform. The method is lightweight and it allows for quick reuse of available models on different simulation or runtime platforms, without the need for sophisticated middleware technologies.

Finally, the runtime computation details of the DEVSRT approach were investigated and an RT task model comprising of the DEVS intrinsic processes has been proposed. This model was used as starting point for further investigation regarding task scheduling and resource management in a DEVS-based RT system. The DEVSRT task model was integrated with IC approach in an innovative method, in which the model behavior is prioritized, allowing for efficient and dynamic task scheduling in the system. The overload management policy introduced in this framework provides an early reaction mechanism to transient overrun situations, saving critical outputs from lateness, preventing catastrophic results in the system.

The outcome of this research enables RT and embedded system designers to adopt an M&S-based approach, bridging the gap between simulation and RT software development. It also opens a new horizon towards model-based operating system design, allowing for creation of systems dedicated to run models as processes.

6.1 Review of the Contributions

This section reviews the major contributions in the two research areas investigated in this dissertation, namely: the DEVSRT simulation-driven development methodology for RT and

embedded systems development and the I-DEVS approach to achieve a hard RT system design scheme by integrating the DEVSRT with IC techniques. The following subsections summarize the key contributions regarding these research objectives.

A) DEVSRT

DEVSRT applies a dynamic DEVS-based approach for embedded real-time application development. It takes advantage of well-defined M&S properties and constructs of DEVS to design and interface embedded systems with their hardware surrogates. The following contributions are the outcome of this approach:

- The use of physical time in the DEVS event scheduling paradigm, enabling the runtime engine to trigger the events based on the clock of the system, providing an RT simulation engine.
- DEVS has been investigated to be adopted as an RT and embedded application development technique; hence the corresponding features of DEVS are highlighted and applied in the design of example models.
- A formal way of defining deadline for DEVS outputs has been proposed and the details of hardware interaction using DEVS have been discussed. DEVS formal I/O ports were used to interface models with hardware or external environment. This method provided the basis for hardware integration and model continuity.
- The interfacing mechanism between DEVS and external environment has been improved and a new version has been adopted for DEVSRT.
- A generic lightweight interface for message transfers between DEVS models running on different DEVS-based tools has been presented. This framework was used to develop and execute shared controller model for robots, emergency management and combat simulations.

- The Embedded CD++ (E-CD++) tool has been extended to implement the DEVSRT approach. The new version of E-CD++ was implemented on Xenomai real-time kernel, incorporating real-time services provided by the kernel.
- The input receiving stubs are implemented as separate Xenomai tasks working concurrently, to receive data from different ports, though not interrupting the main runtime task.
- An Eclipse-based plug-in IDE offering embedded functionalities and graphical model designer capability was provided allowing for rapid design and deployment of the models.
- Several RT embedded systems and controllers have been designed and implemented on a variety of hardware platforms such as FPGAs, embedded boards, and robotic devices.

B) I-DEVS (Imprecise DEVS)

The second major contribution of this dissertation is the adoption of the IC approach with the DEVSRT platform to propose a more reliable model-based design and execution platform for hard real-time systems. The followings were the contributions towards this goal:

- The active processes in an RT DEVS-based system have been identified and a task system has been proposed to be used as a foundation for further investigation and study of RT aspects in these systems.
- The Imprecise DEVS (I-DEVS) formalism has been proposed incorporating IC-based inherent concepts with the DEVSRT framework.
- A scheduling algorithm based on reacting early to the overload scenario has been introduced, which was later demonstrated through experiments.
- The IC concepts have been integrated with DEVS modeling framework, in an abstract way, enabling the modeler to design imprecise models independent of the simulation engine.
- Detailed examples of overload scenarios and the efficiency of the approach have been presented.

- The I-DEVS M&S framework was implemented on E-CD++, providing a development platform for imprecise modeling and execution using DEVS formalism.
- Several test cases measuring the integrity, consistency, and efficiency of the algorithm have been carried out.

6.2 Future Work

The following is a list of possible future research trends in DEVSRT, collaborative RT modeling, and I-DEVS approache:

- Integration of model checking and formal verification techniques with the DEVSRT approach can be investigated in order to have an integrated modeling tool incorporating techniques to discover anomalies, inconsistencies, deadlocks, and other pitfalls in the model.
- Design and development of embedded operating systems can be investigated using DEVSRT as a platform. The OS will autonomously operate by executing the models functioning as processes in the system. This will open a new direction in embedded system design employing MDE approach in the entire design and development phases.
- Design and development of techniques to formally interface DEVSRT with different graphical modeling and visualization tools on embedded platforms towards the creation of interactive virtual reality and simulation-based games.
- Integration of more efficient synchronization protocols can be investigated in the lightweight RT model execution protocol. For example, Lamportian physical clock synchronization [Lam78] using partial and total ordering of events can be explored further.
- Incorporating Dynamic DEVS formalism with the proposed I-DEVS formalism to introduce a new imprecise DEVS capable of prioritizing different components of the model besides the behaviors. This way, the system can shut down an entire optional component in a model in order to open space for the other processes.

- Sensitivity analysis on the I-DEVS and DEVSRT functional parameters (e.g. grace period, inter-thread message transfer delay, …), in order to explore target specific configurations and help system designers tune their system according to the underlying hardware and middleware platform.
- Applying schedulability analysis to the proposed DEVS-based task model in order to determine the system capacity in handling different sizes of processing load. On the other hand it helps the scheduler to determine whether it can accept a task or not. Rate monotonic tests can be a good starting point to measure the schedulability of the model on any specific hardware platform.
- Expansion of the key scheduling and overload detection algorithms in the I-DEVS framework, in order to design more efficient and target-specific hard RT system development tools.
- Investigating approaches to make the run-time engine more efficient and reducing the overhead of model execution. E-CD++ currently support flat coordinator technique, where the control hierarchy is flattened to reduce the message transfer overhead. However, there is a tradeoff between the complexity of the flat coordinator and the reduced overhead. These challenges can be researched to propose recommendations for modelers about the best choice in using flattened or hierarchical coordinator for any specific model.

References

[Abr06] Abrial, J. R., “Formal methods in industry: achievements, problems, future”, Proceeding of the 28th international conference on Software engineering, pp. 761–768, New York, NY, USA, 2006.

[Ada90] “Ada performance issues”, Ada Letters, SIGAda, ACM Press, vol. 10, no. 3, 1990.

[Ahm11] Ahmed, A. S., M. Moallemi, G. Wainer, and S. Mahmoud, “Cell-DEVS & 3D Real-Time Visual Simulation to Support Combat”, Proceedings of Summer Simulation Conference (SCSC'11), Netherland, 2011 (Second Best Paper).

[Ami03] Amirijoo, M., J. S. Hansson, and H. Son, “Error-Driven QoS Management in Imprecise Real-Time Databases”, Proceedings of the 15th Euromicro Conference on Real-Time Systems, pp. 63, Porto, Portugal, 2003.

[Ara08] Araujo, R. B., F. M. Iwasaki, E. B. Pizzolato, A. Boukerche, “A Framework for 3D web-based visualization of HLA-compliant simulations”, Proceedings of the 13th international symposium on 3D web technology, pp. 83-90, Los Angeles, CA, 2008.

[Ayd99] Aydin, H., P. Mejia-Alvarez, R. Melhem, and D. Moss´e, “Optimal Reward-Based Scheduling of Periodic Real-Time Tasks”, IEEE Transactions On Computers, pp. 111-130, 1999.

[Bal97] Balarin, F., et. al. "Hardware-Software Co-design of Embedded Systems. The POLIS Approach", Kluwer Academic Publishers, 1997.

[Bal06] Balasubramanian, K., A. Gokhale, G. Karsai et al., “Developing Applications Using Model-Driven Design Environments”, Journal of COMPUTER, vol. 39, no. 2, pp. 33-40, 2006.

[Bar97] Barros, F. J., “Modeling Formalisms for Dynamic Structure Systems”, ACM Transactions on Modeling and Computer Simulation, vol. 7, no. 1, pp. 501-515, 1997.

[Bar98] Baruah, S., and M. Hickey, “Competitive On-Line Scheduling of Imprecise Computations” IEEE Transaction on Computer, vol. 47, no.9, pp. 1027–1032, 1998.

[Bas06] Basu, A., M. Bozga, and J. Sifakis, “Modeling Heterogeneous Real-time Components in BIP”, Proceedings of 4th IEEE International Conference on Software Engineering and Formal Methods, pp. 3-12, Pune, India, 2006.

[Ber10] Bergero, F., and E. Kofman. “PowerDEVS: A Tool for Hybrid System Modeling and Real-time Simulation”, Journal of SIMULATION, vol. 87, no. 1-2, pp. 113-132, 2010.

[Bon10] Bonaventura, M., G. A. Wainer, R. Castro, “Advanced IDE for Modeling and Simulation of Discrete Event Systems”, Proceedings of 2010 Spring Simulation Conference (SpringSim10), DEVS Symposium, Orlando, FL, 2010.

[Bou97] Boussinot, F., and R. de Simone, “The ESTEREL language”, Proceedings of the IEEE Software Journal, vol. 79, no. 9, pp. 1293-1304, 1991.

[Bou05] Bouyssounouse, B., and J. Sifakis, “Embedded Systems Design: The ARTIST Roadmap for Research and Development”, Lecture Notes in Computer Science 3436, Springer-Verlag, 2005.

[Bou08] Boukerche, A., F. M. Iwasaki, R. B. Araujo, E. B. Pizzolato, “Web-Based Distributed Simulations Visualization and Control with HLA and Web Services”, Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, pp.17-23, Vancouver, BC, Canada, 2008.

[Bou09] Boukerche, A., M. Zhang, and R. Pazzi, “An adaptive virtual simulation and RT emergency response system”, Proceedings of the International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, pp. 360-364, Hong Kong, China, 2009.

[But06] G. Buttazzo, “Achieving Scalability in Real-Time Systems” IEEE Computer, vol. 39, no. 5, pp. 54–59, May 2006.

[But10] Buttazzo, G. C., "Hard real-time computing systems: predictable scheduling algorithms and applications", Springer, Second Edition, 2010, ISBN: 0792399943.

[Cap03] Capocchi, L., F. Bernardi, D. Federici, and P. Bisgambiglia, “Transformation of VHDL descriptions into DEVS models for fault modeling and simulation”, Proceedings of the IEEE Systems, Man and Cybernetics Conference, pp. 1205-1211, Washington, USA, 2003.

[Car93] Carlsson, C., and O. Hagsand, “DIVE: A Multi-user Virtual Reality System”, IEEE Virtual Reality Annual International Symposium, pp. 394-400, Seattle, WA , USA, 1993.

[Cas11] Castro, R., E. Kofman, and F. E. Cellier, “Quantization-based integration methods for delay-differential equations”, Journal of Simulation Modelling Practice and Theory, vol. 19, no. 1, pp. 314-336, 2011.

[Cel06] Cellier, F., and E. Kofman, “Continuous System Simulation”, Springer-Verlag, ISBN: 978-0-387-26102-7, New York, 2006.

[Cha09] Chaturvedi, D. K., "Modeling and Simulation of Systems Using MATLAB and Simulink", CRC Press, 2009, ISBN: 1439806721.

[Che97] Chen, X., and A. M. K. Cheng, “An Imprecise Algorithm for Real-Time Compressed Image and Video Transmission”, Proceedings of 6th International Conference on Computer Communications and Networks, pp. 390-397, Las Vegas, NV., USA, 1997.

[Che09] Chen, J. M., W. C. Lu, W. K. Shih, M. C. Tang, “Imprecise Computations with Deferred Optional Tasks”, Journal of Information Science and Engineering, vol. 25, no. 1, pp. 185-200, 2009.

[Chi07] Chidisiuc, C., and G. Wainer, “CD++Builder: An Eclipse-based IDE for DEVS Modeling”, Proceedings of the 2007 Spring Simulation Multiconference, pp. 235-240, Norfolk, VA, 2007.

[Cho94] Chow, A. C., and B. P. Zeigler, “Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism”, Proceedings of Winter Simulation Conference, pp. 716-722, Orlando, FL, 1994.

[Cho98] Cho, S. M., and Kim T. G. “Real-Time DEVS Simulation: Concurrent, Time-Selective Execution of Combined RT-DEVS Model and Interactive Environment”, Proceeding of Summer Simulation Conference, pp. 410-415, Reno, Nevada, 1998.

[Cho00] Cho, Y. K., B. P. Zeigler, H. J. Cho, et al., “Design Considerations for Distributed Real-Time DEVS” Proceedings of AI, Simulation and Planning Conference, Tucson, Arizona, 2000.

[Cho03] Cho, Y. K., 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.

[Chr04] Christen, G., A. Dobniewski and G. Wainer, “Modeling State-Based DEVS Models in CD++”, Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04). Arlington, VA. U.S.A

[Chu90] Chung, J. Y., J.W. S. Liu, and K. J. Lin, “Scheduling Periodic Jobs That Allow Imprecise Results”, IEEE Transaction on Computer, vol. 39, no9, pp. 1156–1174, 1990.

[Cor01] Cortelessa, V., A. D’Ambrogio, and G. Iazeolla, “Automatic Derivation of Software Performance Models from Case Documents,” Journal of Performance Evaluation, vol. 45, no. 2-3, pp. 81-105, 2001.

[D’Am05] D’Ambrogio, A., “A Model Transformation Framework for the Automated Building of Performance Models from UML Models”, Proceedings of the ACM Fifth International Workshop on Software and Performance, pp. 75-86, Palma de Mallorca, Spain, 2005.

[Eke03] Eker, J., J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong, “Taming heterogeneity the Ptolemy approach”, Proceedings of the IEEE Transaction, vol. 91, pp. 127-144, 2003.

[Fen93] Feng, W., and J. W. S. Liu, “An Extended Imprecise Computation Model for Time-Constrained Speech Processing and Generation”, Proceedings of the IEEE Workshop on Real-Time Applications, pp. 76 – 80, New York, NY., USA, 1993.

[Fin96] Finney, K., “Mathematical notation in formal specification: too difficult for the masses”, IEEE Transactions on Software Engineering, vol. 22, no.2, pp.158–159, 1996.

[Fuj99] Fujisawa, K., S. Hayakawa, T. Aoki, T. Suzuki, and S. Okuma, “Real Time Motion Planning for Autonomous Mobile Robot, using Framework of Anytime Algorithm” Proceedings of the IEEE International Conference on Robotics & Automation, pp. 1347-1352, Detroit, Michigan, USA, 1999.

[Gia03] Giambiasi, N., J. L. Paillet, and F. Châne, “Simulation and verification II: from timed automata to DEVS models”, Proceedings of the Winter Simulation Conference, pp. 923–931, Louisiana, USA, 2003.

[Gli02] Glinsky, E., and G. Wainer, “Performance analysis of real-time DEVS models”, Proceedings of the Winter Simulation Conference, pp. 588–594, San Diego, CA, 2002.

[Gli04a] Glinsky, E., and G. Wainer, “Modeling and simulation of systems with hardware-in-the-loop”, Proceedings of Winter Simulation Conference, Washington D.C, 2004.

[Gli04b] Glinsky, E., and G. Wainer, “Model-Based Development of Embedded Systems with RT-CD++”, Proceedings of the WIP session, IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, ON, Canada, 2004.

[God07] Godding, G., H. Sarjoughian, and K. Kempf, “Application of Combined Discrete-event Simulation and Optimization Models in Semiconductor Enterprise Manufacturing Systems”, Proceedings of the Winter Simulation Conference, pp. 1729-1736, Washington D.C., 2007.

[Gro02] Grotker, T., S. Liao, G. Martin, and S. Swan, “System Design with SystemC”, Kluwer Academic Publishers, Netherlands, 2002.

[Han00] Hansson, J., M. Thuresson, and S. Son, “Imprecise Task Scheduling and Overload Management using OR-ULD”, Proceedings of 7th International Conference on Real-Time Computing Systems and Applications, pp. 307-314, Cheju Island, South Korea, 2000.

[Har08] Harzallah, Y., V. Michel, Q. Liu, and G. Wainer, “Distributed Simulation and Web Map Mash-Up for Forest Fire Spread”, Proceedings of the 2008 IEEE Congress on Services – Part I, pp. 176-183, Honolulu, HI, 2008.

[Hil08] Hill, F.S., and S.M. Kelley, “Computer Graphics using OpenGL”, Prentice Hall publishers, ISBN: 0131496700, 3rd Edition, 2008.

[Hol09] Holman, K., J. Kuzub, M. Moallemi, G. A. Wainer, “Cable-Anchor Robot Implementation using Embedded CD++”, Poster in proceedings of SIMUTools Conference, Rome, Italy, 2009.

[Hon97] Hong J. S., Song H. H., Kim T. G., and Park K. H., “A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development”, Springer Netherlands, 1997.

[Hu01] Hu, X., B. P. Zeigler, and J. Couretas, “DEVS-On-A-Chip: Implementing DEVS In Embedded Java On A Tiny Internet Interface For Scalable Factory Automation”, Proceedings of the IEEE Systems, Man, and Cybernetics Conference, pp. 3051-3056, Tucson, AZ , USA, 2001.

[Hu04] Hu, X., and B. P. Zeigler, “Model Continuity to Support Software Development for Distributed Robotic Systems: a Team Formation Example”, Journal of Intelligent & Robotic Systems, Theory & Application, vol. 39, no. 1, pp. 71-87, 2004.

[Hu05] Hu, X., and B. P. Zeigler, “Model continuity in the design of dynamic distributed real-time systems”, IEEE Transactions on Systems, Man and Cybernetics, Part A, vol. 35, no. 6, pp. 867-878, 2005.

[Hu07] Hu, W., and H. Sarjoughian, “A co-design modeling approach for computer network systems”, Proceedings of the 39th Winter Simulation Conference, pp. 685–693, Washington D.C., 2007.

[Hua95] Huang, X., and A. M. K. Cheng, “Applying Imprecise Algorithms to Real-Time Image and Video Transmission” Proceedings of Real-Time Technology and Applications Symposium, pp. 390, Chicago, Illinois, USA, 1995.

[Hua04] Huang, D., and H. Sarjoughian, “Software and Simulation Modeling for Real-Time Software-Intensive Systems”, In Proceedings of 8th IEEE Symposium on Distributed Simulation and Real-time Applications, pp. 196-203, Budapest, Hungary, 2004.

[Hua06] Huang, D., H. Sarjoughain, G. Godding et al., “Flexible experimentation and analysis for hybrid DEVS and MPC models”, Proceedings of the 38th Winter Simulation Conference, pp. 1863-1870, Monterey, CA, USA, 2006.

[IEE10] IEEE standard for “Modeling and Simulation (M&S) High Level Architecture (HLA) Framework and Rules”, IEEE Std. pp. 1516-2010, 2010.

[Jac02] Jacques, C., and G. Wainer, “Using the CD++ DEVS toolkit to develop Petri Nets”, Proceedings of Summer Computer Simulation Conference, San Diego, CA. USA. 2002.

[Jaf10] Jafer, S., and G. A. Wainer, “Conservative DEVS - A Novel Protocol for Parallel Conservative Simulation of DEVS and Cell-DEVS Models”, Proceedings of Spring Simulation Conference, DEVS Symposium, pp. 168-175, Orlando, FL., 2010.

[Kar89] Kar, R. P., and K. Porter, “Rhealstone, a real time benchmark proposal; an independently verifiable metric for complex multitaskers”, Dr. Dobb’s Journal,1989.

[Kar90] Kar, R. P., “Implementing the Rhealstone real-time benchmark, where a proposal’s rubber meets the real-time road”, Dr. Dobb’s Journal, April 1990.

[Kim01] Kim, J. K., Y.G. Kim, and T.G. Kim, “DHMIF: DEVS-Based Hardware Model Interchange Format”, Proceedings of European Simulation Symposium, Marseille, France, 2001.

[Kim04] Kim, K. H., and W. S. Kang, “CORBA-Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Nonhierarchical

One", Proceedings of the 2004 International Conference on Computational Science and Its Applications, Assisi, Italy, pp. 167-176, 2004.

[Kob04] Kobayashi, H., and N. Yamasaki, "RT-Frontier: A Real-Time Operating System for Practical Imprecise Computation", Proceedings of the 10th IEEE Real-Time and Applications Symposium, pp. 255-264, Toronto, Canada, 2004.

[Kus01] Kuster, J., and J. Stroop, "Consistent Design of Embedded Real-Time Systems with UML-RT", Proceedings of 4th Int. Symp. on Object-Oriented Real-Time Distributed Computing, pp. 31-40, Magdeburg , Germany, 2001.

[Lam78] Lamport, L. "Time, clocks, and the ordering of events in a distributed system", Communications of ACM, vol. 21, no. 7, pp. 558-565. 1978

[Led01] Lédeczi, Á., Á. Bakay, M. Maróti et al., "Composing Domain-Specific Design Environments," Journal of COMPUTER, vol. 34, no. 11, pp. 44-51, 2001.

[Li03] Li, L., T. Pearce, and G. Wainer, "Interfacing Real-time DEVS models with a DSP platform", Proceedings of the Industrial Simulation Symposium, Valencia, Spain, 2003.

[Liu73] Liu, C. L., and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of ACM, vol. 20, no. 1, pp. 46-61, 1973.

[Liu91] Liu, J. W. S., K. J. Lin, W. K. Shih J. Y. Chung, A. Yu, and W. Zhao, "Algorithms for Scheduling Imprecise Computations", IEEE Transaction on Computer, vol. 24, no. 5, pp. 58-68, May 1991.

[Liu94a] Liu, J. W. S., W. Shih, K. J. Lin, R. Bettati, and J. Chung, "Imprecise Computations", Proceedings of the IEEE, vol. 82, no.1, pp. 83–94, 1994.

[Liu94b] Liu, J.W.S., K. J. Lin, R. Bettati, D. Hull, and A. Yu. "Use of imprecise computation to enhance dependability of real-time systems", The International Series in Engineering and Computer Science, vol. 284, no. 3, pp.157-182, 1994.

[Liu95] Liu, J. W. S., and W. K. Shih, "Algorithms for Scheduling Imprecise Computations with Timing Constraints to Minimize Maximum Error", IEEE Transaction on Computer, vol. 44, no. 3, pp. 466–471, 1995.

[Liu00] Liu, J. W. S., "Real-Time Systems" Upper Saddle River, NJ: Prentice-Hall, 2000, ISBN: 0-13-099651-3.

[Liu07] Liu, Q., and G. Wainer, “Parallel Environment for DEVS and Cell-DEVS Models”, Journal of SIMULATION, vol.83, no.6, pp. 449-471, 2007.

[Lom06] Lombardi, S., G. Wainer, and B. P. Zeigler, “Interoperation of DEVS models in DEVS/C# and CD++” Proceedings of SISO Fall Interoperability Workshop, Huntsville, AL, 2006.

[Mat11] The MathWorks website: http://www.mathworks.com, visited July 2011.

[Mit09] Mittal, S., J. L. Risco-Martin, and B. P. Zeigler, “DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process”, SIMULATION, vol. 85, no.7, pp. 419-450, 2009.

[Moa08] Moallemi, M., M. Alcaraz, and G. Wainer, “ECD++ A DEVS based Real-Time Simulator for Embedded Systems”, Poster in proceedings of Spring Simulation Conference, Ottawa, Canada, 2008.

[Moa09] Moallemi, M., and G. A. Wainer, “A System-On-Chip FPGA Implementation of Embedded CD++”, Proceedings of Spring Simulation Conference, San Diego, CA, USA, 2009.

[Moa10a] Moallemi, M., and Gabriel Wainer, “A Simplified Real-Time Embedded DEVS Approach Towards Embedded and Control Design”, Poster in proceedings of Winter Simulation Conference, Austin, USA, 2010.

[Moa10b] Moallemi, M., and G. A. Wainer, “Designing an Interface for Real-Time and Embedded DEVS”, Proceedings of Spring Simulation Conference, DEVS Symposium, Orlando, Florida, USA, 2010.

[Moa10c] Moallemi, M., D. A. Tall, G. A. Wainer, and A. Awad, “Application of RT-DEVS in Military”, Proceedings of Spring Simulation Conference, MMS Symposium, Orlando, Florida, USA, 2010.

[Moa11a] Moallemi, M., R. Castro, F. Bergero, and G. A. Wainer, “Component-Oriented Interoperation of Real-Time DEVS Engines”, Proceedings of Spring Simulation Conference, ANSS Symposium, Boston, MA, USA, 2011.

[Moa11b] Moallemi, M., S. Jafer, A. S. Ahmed, and G. Wainer “Interfacing DEVS and Visualization Models for Emergency Management”, Proceedings of Spring Simulation Conference, Work In Progress of the DEVS Symposium, Boston, MA, USA, 2011.

[Mon03] Monin, J. F., and M. G. Hinchey, “Understanding formal methods”, Springer, 2003, ISBN: 1852332476.

[Mon09] Mondada, F., M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, “The e-puck, a robot designed for education in engineering”, Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions, pp. 59–65, Castelo Branco, Portugal, 2009.

[Neu66] Neumann, J. V., and A. W. Burks, “Theory of Self-Reproducing Automata”, Champaign: University of Illinois Press, 1966.

[Nic10] Nicolescu, G., and P. J. Mosterman “Model-Based Design for Embedded Systems”, CRC Press 2010, ISBN: 978-1-4200-6784-2.

[Pan96] Pandzic, I., T. Capin, N. Magnenat-Thalmann, and D. Thalmann, “Towards Natural Communication in Networked Collaborative Virtual Environments”, Proceedings of FIVE Conference, Framework for Immersive Virtual Environments, pp. 37-47, 1996.

[Par02] Parker, G. B., “Punctuated Anytime Learning for Hexapod Gait Generation” Proceedings of IEEE/RSJ International Conference on Intelligent Robots and System, vol. 3, pp. 2664–2671, Beijing, China, 2002.

[Pra93] Praehofer, H., and D. Pree, “Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs”, Proceedings of the Winter Simulation Conference, pp.595-603, Los Angeles, CA, 1993.

[Saa09] Saadawi, H., and G. Wainer, “Verification of real-time DEVS models”, Proceedings of DEVS Symposium, San Diego, CA. 2009.

[Saa11] Saadawi, H., G. Wainer, and M. Moallemi, “Principles of DEVS Models Verification for Real-Time Embedded Applications” chapter in the book “Real-time Simulation Technologies: Principles, Methodologies, and Applications”, Pieter Mosterman and Katalin Popovici, CRC Press, 2011.

[Sad10] Sadeghi, F. R., G. Wainer, and M. Moallemi “Modeling and Controlling a Robotic Arm with E-CD++”, Poster in proceedings of Summer Simulation Conference, Ottawa, ON, Canada, 2010.

[Sag04] Saghir, A., T. Pearce, and G. Wainer, “Modeling Computer Hardware Platforms using DEVS and HLA Simulation,” SIMULATION SERIES, vol. 36, no. 4, pp. 218, 2004.

[Sar87] Sargent, R.G., “An Overview of Verification and Validation of Simulation Models”, Proceedings of the Winter Simulation Conference, New York, NY, USA, 1987.

[Sar98] Sarjoughian, H., and B. P. Zeigler, “DEVSJAVA: Basis for a DEVS-based collaborative M&S environment”, Proceedings of the International Conference on Web-based Modeling & Simulation, pp. 29-36, San Diego, CA, 1998.

[Sar99] Sarjoughian, H., J. Nutaro, and B. P. Zeigler, “Collaborative DEVS Modeler” Proceedings of the International Conference on Web-Based Modeling and Simulation, San Francisco, CA, 1999.

[Sar00] Sarjoughian, H., and B. P. Zeigler, “DEVS and HLA: Complimentary Paradigms for M&S”, Transactions of the SCS Organization, vol. 17, no. 1, pp. 187-197, 2000.

[Sar01a] Sarjoughian, H., X. Hu, D. Hild et al., “Simulation-based SW/HW Architectural Design Configurations for Distributed Mission Training Systems”, Journal of SIMULATION, vol. 77, no. 1/2, pp. 23-38, 2001.

[Sar01b] Sarjoughian, H., S. Park, and B. P. Zeigler, “Collaborative distributed network system: a lightweight middleware supporting collaborative DEVS modeling”, Future Generation Computer Systems vol. 17, no. 1, pp. 89–105, 2001.

[Sch00] Schulz, S., T.C. Ewing, and J.W. Rozenblit, “Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling”, Proceedings of IEEE International Conference on the Engineering of Computer Based Systems, pp. 308-308, Edinburgh, UK, 2000

[Sel01] Selic, B., “The emerging real-time standard [UML]”, Proceedings of 6th international Workshop Object-Oriented Real-Time Dependable Systems, pp. 3-9, Rome, Italy, 2001.

[Sha07] Shang, H., and G. A. Wainer, “A flexible dynamic structure DEVS algorithm towards embedded systems”, Proceedings of the Summer Computer Simulation Conference, pp. 339-345, San Diego, California, 2007.

[Shi91] Shih, W. K., J. W. S. Liu, and J. Y. Chung, “Algorithms for Scheduling Imprecise Computations with Timing Constraints”, SIAM Journal of Computer, vol. 20, no. 3, pp. 537–552, 1991.

[Shi92] Shih, W. K. and J. W. S. Liu, “On-Line Scheduling of Imprecise Computations to Minimize Total Error”, Proceedings of the 13th IEEE Real-Time Systems Symposium, Phoenix, Arizona, pp. 280-289, 1992.

[Shi96] Shih, W. K., and J. W. S. Liu, “On-line algorithms for scheduling imprecise computations”, SIAM Journal on Computing, vol. 25, no. 1, pp. 1105-1121, 1996.

[Son05] Song, H. S., 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.

[Tra06] Travis, J., and J. Kring, “LabVIEW for Everyone: Graphical Programming Made Easy and Fun”, 3rd Edition, Prentice Hall, 2006, ISBN: 0-13-185672-3.

[Tro03] Troccoli, A., and G. Wainer, “Implementing Parallel CD++”, Proceedings of the Annual Simulation Symposium, Orlando, FL. 2003.

[Veg11] Vega Prime software page on Presagis corporation website, available at: http://www.presagis.com/products_services/products/ms/visualization/vega_prime, accessed July 2011.

[Wai02a] Wainer, G., and N. Giambiasi, “N-dimensional Cell-DEVS Models”, Discrete Event Dynamic Systems, vol. 12, no. 2, pp. 135-157, 2002.

[Wai02b] Wainer, G., “CD++: A Toolkit to Develop DEVS Models”, Software – Practice and Experience, vol.32, no.13, pp. 1261-1306, 2002.

[Wai04] Wainer, G., and E. Glinsky, "Model-Based Development of Embedded Systems with RT-CD++", Proceedings of the WIP session, IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, ON., Canada, 2004

[Wai05] Wainer, G., E. Glinsky, P. MacSween “A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism”. Model-driven Software Development, Vol. 2 of Research and Practice in Software Engineering. S. Beydeda and V.Gruhn Eds. Springer-Verlag. 2005.

[Wai08a] Wainer, G., Q. Liu, J. Chazal, L. Quinet, and M. K. Traore, “Performance Analysis of Web-based Distributed Simulation in DCD++: A Case Study across the Atlantic Ocean”, Proceedings of the 2008 Spring Simulation Multiconference: High Performance Computing Symposium, pp. 413-420, Ottawa, Canada, 2008.

[Wai08b] Wainer, G., R. Madhoun, and K. Al-Zoubi, “Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web-Services”, Simulation Modeling Practice and Theory, 16(9), pp. 1266-1292, 2008.

[Wai09] Wainer, G. A., "Discrete-event modeling and simulation; a practitioner's approach", CRC / Taylor & Francis, ISBN: 9781420053364, 2009.

[Wai11] Wainer, G., and R. Castro “DEMES: a Discrete-Event methodology for Modeling and simulation of Embedded Systems”, Accepted in Modeling and Simulation Magazine, Society for Modeling and Simulation International, San Diego, CA., 2011.

[Wan03] Wang, Y. and Y. Liao, “Implementation of a Collaborative Web-based Simulation Modeling Environment”, Proceedings of the Seventh IEEE Workshop on Distributed Simulation and Real-Time Applications, pp.150-157, 2003.

[Wat97] Waters, R., D. Anderson, J. Barrus, D. Brogan, M. Casey, S. Mckeown, T. Nitta, I. Sterns, and W. Yerazunis, “Diamond Park and SPLINE: Social Virtual Reality with 3D Animation, Spoken Interaction and Runtime Extendability Presence”, Journal of Teleoperators and Virtual Environments, vol. 6, no. 4, pp. 461–481, 1997.

[Wie08] Wiedenhoft, G. R., and A.A. Fröhlich. “Using Imprecise Computation Techniques for Power Management in Real-Time Embedded Systems”, Proceedings of 6th IFIP Working conference on Distributed and Parallel Embedded Systems, pp. 121-130, Milano, Italy. 2008.

[Xen11] Xenomai Real-Time Kernel for Linux: www.xenomai.org, visited July 2011.

[Xns11] Xenomai Native Skin Functions User Reference: www.xenomai.org/documentation/branches/v2.3.x/pdf/Native-API-Tour-rev-C.pdf, visited July 2011.

[Yu07a] Yu, Y. H., and G. Wainer, “eCD++: an engine for executing DEVS models in embedded platforms” Proceedings of the 2007 SCS Summer Computer Simulation Conference, San Diego, CA, USA, pp. 323-330. 2007

[Yu07b] Yu, Y. H., “Designing extensions for the use of CD++ to build embedded discrete-event systems”, Master thesis submitted to Systems and Computer Engineering Department, Carleton University, 2007.

[Zei93] Zeigler, B. P., and J. Kim, “Extending the DEVSScheme Knowledge-Based Simulation Environment for Real-Time Event-Based Control”, IEEE Transaction On Robotics and Automation, vol. 9, no. 3, pp. 351-356, 1993.

[Zei96] Zeigler, B. P., Y. Moon, D. Kim, and J. G. Kim, “DEVS-C++: A High Performance Modelling and Simulation Environment”, Proceedings of the 29th Annual Hawaii International Conference on System Sciences, Maui, HI, pp. 350-359, 1996.

[Zei00] Zeigler, B., T. Kim, and H. Praehofer, “Theory of Modeling and Simulation”, Academic Press, ISBN: 0127784551, 2000.

[Zei03] Zeigler, B. P., “DEVS Today: Recent Advances in Discrete Event-Based Information Technology”, Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, Orlando, FL, pp. 148-161, 2003.

[Zhe03] Zheng, T., and G. Wainer, “Implementing finite state machines using the CD++ toolkit”, Proceedings of the SCS Summer Simulation Conference, Montreal, Canada, 2003.

[Zil93] Zilberstein, S., and S. J. Russel. “Anytime Sensing, Planning and Action: A Practical Model for Robot Control” Proceedings of the 13th International Joint Conference on Artificial Intelligence, pp. 1402-1407, Chambery, France, 1993.