An Environment for Development and Benchmarking DEVS applications

By

J. Marcelo Gutierrez-Alcaraz

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment for the degree of
Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

Canada

Your file Votre référence
ISBN: 978-0-494-33651-9
Our file Notre référence
ISBN: 978-0-494-33651-9

Abstract

This thesis deals with the construction of an integrated development and benchmarking environment for models developed using the Discrete Event System Specification (DEVS), which is a formal modeling and simulation (M&S) framework that supports hierarchical, modular models. DEVS-based M&S environments have been used successfully to understand, analyze, and develop a variety of systems. The tool used to construct this environment is based on the CD++ toolkit, which is designed as a DEVS model and offers great flexibility when building, testing and debugging DEVS models.

The first part deals with the design of a graphical Integrated Development Environment (IDE) interface, developed to provide a major speed up from the conception to the construction and testing of models of Real-Time embedded systems; it also helps the designer produce and test the final implementation code and deployment on an embedded target. A description of a hardware-in-the-loop example using the IDE with the embedded version of CD++, and the computer's parallel port is shown; this example is used to demonstrate the gain in development time provided by the new IDE tool interface.

The second part of the thesis deals with the adaptation and modification of a performance benchmarking tool, based on the Dhrystone Benchmark. The benchmarking tool, called DEVStone, can be used for the comparison of the performance of different versions of CD++ against each other or with different DEVS-based simulators. For this last case an example is presented and a testing methodology is introduced with the use of the ADEVS (A Discrete Event Simulator) simulator, which is another DEVS-based simulator.

ACKNOWLEDGEMENTS

I would like to thank the invaluable supervision and support of Dr. Gabriel Wainer during the development of this work.

To my parents Lucio and Sonia and my siblings: Claudia, Christian and Daniela, their constant support and love through the course of my studies took me where I am.

My wholehearted thanks go to Pamela and Jeremy Clark, my "adoptive" family in Ottawa.

Finally yet importantly, to my friends and colleagues in the group for their support and help answering my questions.

Table of Contents

Table of Figures

Table of Acronyms

ADEVS	A Discrete EVent System
DDR	Double Data Rate
DEVS	Discrete EVent System Specification
ECD++	Embedded CD++
EIC	External Input Couplings
EOC	External Output Couplings
ETF	External Transition Function
FSB	Front Side Bus
GCC	GNU Compiler Collection
GUI	Graphical User Interface
HIL	Hardware In the Loop
IC	Internal Coupling
IDE	Integrated Development Environment
IP	Internet Protocol
ITF	Internal Transition Function
M&S	Modeling and Simulation
NFS	Network File System
OOM	Out Of Memory
OS	Operating System
PERL	Practical Extraction and Report Language
RAM	Random Access Memory
RT	Real Time
TMO	Time-triggered Message-triggered Object
TTL	Transistor-Transistor Logic
UML	Unified Modeling Language
XML	eXtensible Markup Language

1. Introduction

Different technologies of modeling and simulation are widely used in the industry and the academy to assist system development. Using abstract models in simple and complex simulations of most process greatly reduces the development time and significant savings in resources and cost are made. Reducing the development time also helps the design of safer systems and environmental-friendly products, since it is possible to test more scenarios and run simulations on each and every one of them. It is because of this that Modeling and Simulation techniques have become an important part of system analysis and later design through history. Mathematical models can be defined as abstract representations of natural events, for engineers and scientist these models usually represent different types of phenomena that can be physical, chemical, economical, and social or many others. And by Simulation (by Computer Simulation in particular), we understand it as the process that takes those abstract mathematical models, and through a controlled update of certain defining variables, evolves those models to a different state.

Commonly, the simulation is done through simulation tools that are used at different stages of system development: the analysis phase to support concept development (i.e. virtual prototyping) and in the implementation and test stages to provide virtual test environments (via hardware-in-the-loop techniques) and experimental scenarios for system verification and evaluation [1]. By using abstract models (which depend on the simulation tool used) of real systems in the analysis stage, simulation-based design can highlight problems early enough in the product development process, which in turn may be addressed more cost-effectively on the production side. Many leading companies, among them Boeing, and General Dynamics, have saved millions of dollars on fighter planes, and submarines by replacing physical prototypes with computer mock-ups [2]. Simulation-based test and

verification enable automated test program and test case generation, functional coverage and checking, etc. This virtual test methodology has been widely used, although still in an ad-hoc way, by both hardware and software developers. For example, test generation techniques, tools, and solutions are widely recognized as the main means for hardware verification of complex designs. The approach of using simulation-based software design and implementation combined with hardware-in-the-loop simulation techniques greatly accelerate the embedded software development and integration processes. The effective use of these techniques will result in a faster product development cycle, lower development costs, and higher overall product quality [3].

One particular use of modeling and simulations tools is in the development of embedded systems, usually these systems also have time constraints in which case they are also called Real-Time Systems. Real-Time Systems must provide reliable outputs to external inputs within a time limit. Depending on the strictness of the time limit, the systems are usually separated in soft or hard real time systems [4]. Another characteristic of embedded systems, is that most of them are application specific, although with the increase in computational power from microprocessors this trend is somewhat changing [5]; many of these systems also have a low electrical power constraint because they are deployed in environments where grid-electricity is not commonly available or it is scarce, i.e. inside cars, space ships or remote sensors and actuators.

Many development methods and techniques exist for the creation of embedded systems, with the common denominator that most of them are based on hardware and software that exceeds the computational power of the intended system to be developed. The most common developing system is given by a general-purpose computer, a general-purpose operative system, the target software, which often includes a simulator, and required hardware to communicate with the embedded platform.

For engineering in particular, Modeling and Simulation (M&S) of embedded systems is of utmost importance. For example, engineers and scientists make heavy use of simulation tools when a process is difficult to replicate (because of the cost involved, or if the environmental conditions for the experiment are difficult to replicate or the danger is too high) or when the simulation of a natural process is many times faster than the real process. By using different techniques for modelling, we can predict the behaviour of simple or complicated phenomena with, most of the time, a high degree of certainty. For systems that interact with real data, the preferred method for modeling is the use of continuous differential equations. However, one layer higher in the interaction between systems and the real world we deal with a different nature of modelling and control which is usually easy to model using discrete event modeling methods.

1.1. Background

Zeigler in [6] explains a general framework for an M&S process, and defines the basic entities and their relationships; the basic entities of the M&S process are comprised of a real or virtual environment under analysis; an *experimental frame*, which defines the type of data obtained and the conditions of the system; the model, as stated is an abstract representation of the system to be simulated and; the simulator, which is any computational system capable of executing the model to generate or predict its behaviour. In any M&S framework, it is important to separate the model from the simulator, because this separation between them allows us:

- to reuse a single model for different purposes and,

- to validate and verify both the simulator and the model for easiness of use; (i.e. once the simulator is validated and verified we can assume that the simulator is valid for *any* simulation that we want to run on it as long as the model is valid as well).

In particular, M&S tools have been useful for the development of embedded systems. Since the beginning of the electronic era, most of the capabilities of embedded systems were put on hardware most of the development was done on expensive prototypes. However, with the advent of more powerful microprocessors and the economy of building digital hardware compared to analog hardware, the implementation of their functionalities has steadily shifted to software. This is driven by the fact that software has much more flexibility to cope with system varieties and requirement changes. Recent studies indicate that up to 60% of the development time of an embedded real-time system is spent in software coding [7], [8], [9]. This indicates to us that the existing software development methods are insufficient to develop real-time systems. Actually, the lack of good design methods and support tools has made the software development for embedded systems a bottleneck, especially when a large number of subsystems and task synchronization are involved.

The embedded software developer faces several unique challenges beyond those of classical software development. First, in the case the system is real-time it needs to meet both timeliness and reliability requirements and second the system have constrained resources in terms of memory and processing power. These requirements add extra complexity to the software design and test. For example, for hard real-time systems, special test and analysis techniques have to be adapted or, ultimately, developed to test the correctness of specific control models and to guarantee the system can meet deadlines under all conditions.

In addition, embedded systems usually operate in constantly changing environments, in which the environment itself may be unknown during the design time or it could be continuously evolving as time passes. Therefore, the software that controls these systems should be able to deal with uncertainties, i.e. it could have to reconfigure itself dynamically to adapt to a changing environment. This poses great challenges to test the software effectively under development.

Something else to consider is that the rapid growth of real-time embedded systems brings two other factors into embedded software's complexity. First, embedded systems are making heavy use of networking technologies, among themselves or between them and wired/wireless access points. In the near future, it will be usual for hundreds of embedded controllers, smart sensors and actuators to work together to finish a common task. Consequently, scalability, which was not even considered for this type of systems some time ago, is becoming an important design issue to deal with. Second, with the rapid adoption of cheaper and powerful microprocessors, embedded systems are expected to carry out more and more complex functionalities. It has been foreseen that the new breed of embedded systems (which have enough computational power and memory to carry out complex functionalities) will become dominant [10], making it little practical, if not impossible, to develop physical prototypes in every step of the development process. In order to handle the complexity of these systems, much effort has to be put on system modeling and simulation during the concept proposal, design, analysis, and verification steps.

Historically the state-of-the-art in embedded software development involves a great deal of empirical knowledge and previous experience with particular platforms. Along the time, various efforts to systematize and generalize this approach have been proposed. However, so far none of them fits very well in supporting the design, test, and execution of embedded

software from a systematic way. A compilation of some deficiencies of current development methods is provided in [11]:

- In the software development lifecycle, most of the time different stages are not related to each other, resulting in inconsistencies among analysis, design, test, and implementation. For instance, in the analysis stage of complex systems, mathematical models are usually built to analyze the control algorithms. However, these mathematical models are rarely effectively used by the design stage, which uses different modeling languages such as Unified Modeling Language (UML). The same happens in the implementation stage, which uses programming languages such as C or Java. Because of this constant changing of design and development environments, transformation from one type of model to another is needed among different stages. However, it is important to note that some tools have been developed and improved particularly in the commercial arena, for instance Rational Rose Real-Time sold by International Business Machines (IBM) and the Telelogic family of software development tools. Both systems provide a framework for code generation based on model specification using UML techniques for modeling and some other tools for simulation. These toolkits focus on system model analysis and design and allow graphical description of the system using use-case models and scenarios, activity charts, control block diagram and state-charts. Both environments provide support for maintaining consistency among these models, as well as providing model-driven development environment for software engineering. The formal languages of activity charts and state-charts enable the models' execution and verification using mapping rules. Additionally these products offer, and can produce, graphical interfaces for the project being developed, since it utterly beneficial for software design stage [12].

- Software test for embedded systems is largely ad-hoc and low level. Although control algorithms can be developed and tested in the analysis stage, once they are transformed into implementation code, extensive test is still required because of the discontinuity problem already mentioned. For this reason, many tests are meaningful only after the actual code is generated, and often enough, these tests are meaningful only when the software has been deployed to the real hardware. This low-level activities result in later detection of inconsistencies between the final implementation and the original system specification.

- Despite the continuous need for software to reconfigure itself dynamically in order to adapt to new situations or new environments, "there is no effective and systematic way to design and analyze these kinds of self-adaptive software" [11]. As embedded systems usually operate in real environments, most of them tend to exhibit dynamic reconfiguration to change their structures and operation modes in response to different situations. Hence, it is desirable for an embedded software development method to provide a systematic way to analyze dynamic reconfiguration of systems.

- Scalability becomes an important design specification as embedded systems increasingly work in ad-hoc networks. To ensure scalability, component based technology [13] and suitable software structures and physical topologies are needed. Meanwhile, computer-based modeling and simulation (M&S) methodologies are required since the scale of systems is well beyond what analytical tools alone can handle and it is not always possible to replicate the environments where controlled real experiments could be setup.

To overcome the problems posed by the different models used in different stages of development, the best solution is to provide a *formal method* during the development process. A formal method in this context refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. Where specifications used in formal methods are well-formed statements in a mathematical logic and that, the formal operations are rigorous deductions in that logic (i.e. each step follows from a rule of inference and therefore can be checked by a mechanical process).

The solution proposed in this thesis deals with the use of a formal method, the Discrete Event System Specification (DEVS) formalism, as a basis for the construction of embedded system models with the help of an Integrated Development Environment (IDE) tool, where the developer can find the required tools to continue with the next design stage from just one model. The work done in the thesis provides the tools required for the continuous use of a single model throughout the development process, from the conception of the problem to the implementation in an embedded target. In order to do so, we made use of the CD++ Builder toolkit environment, the embedded version of CD++ [14] and other communications tools to create one development environment as a solution. This solution allows the development of the model, the consequent test and verification through simulation of such model, the development of a control strategy for the variables that need to be controlled by the control system, which can be thoroughly tested and verified, and the deployment of the final code to the embedded system in charge of the control. The proposed solution includes a test case that includes the use of hardware components in the simulation.

Since the use of simulation tools was successfully applied in such a variety of applications due to the ease of model definition, improved composition and reuse, and hierarchical coupling many different simulation tools have arisen. Due to its discrete nature, DEVS

provides considerable precision and speedups in the execution time, as models advance triggered by instantaneous asynchronous events in contraposition with time stepped approaches [15]. The CD++ tool, which is based on the DEVS formalism, allows the rapid development of models and their simulation. However, different versions of this tool have been developed with many improvements and for different purposes and platforms. With every new version, many new features are added to the toolkit but, at the same time, it becomes increasingly more complex to keep track of the impact of the changes, or additions, in the general performance of the M&S toolkit.

To measure the impact on the performance of the simulator and to generate a common metric among different implementation of DEVS simulators, DEVStone [16] was developed. DEVStone is a synthetic model generator that uses the Dhrystone Benchmark as a basic real-time metric. To provide uniform means for obtaining meaningful measurements, the benchmark is based on a large pool of models with different size, complexity and behaviour, resembling different kinds of complex applications. Hence, it is possible to analyze the efficiency of a simulation engine in relation to the characteristics of a category of models of interest. The tool can be used to assess the efficiency of several DEVS simulation engines, and it provides a common metric to compare the results using different tools.

1.2. Contributions

One of the contributions of this Thesis is the development of a simulation-based Integrated Development Environment (IDE) to manage the complexity of developing embedded software. This Integrated Development Environment, based on the DEVS framework, with a front-end based on Eclipse, provides a smooth transition for the developer to design and

test embedded systems on general-purpose computing environments by emphasizing the use and reuse of a single model through the development process. Specifically, this IDE has been developed so that any control models designed and fully tested in multiple simulations, can be deployed, retested and analyzed by emulation in a particular development target.

To improve the actual software testing procedure of the Embedded CD++ (ECD++) tool (where simulations are run in a virtual environment), a new functionality is provided to allow the embedded target to be connected to the real world through sensors and actuators. Consequently, any virtual simulation ran with ECD++ can now be run in real mode with hardware-in-the-loop. This allows analyzing and validating the control algorithms, or to emulate the response of the developed system to a certain event through real actuators.

The Eclipse-based front-end of the CD++ Builder toolkit was populated with the required functionality to create complex discrete event models according to the CD++ language and, if necessary, create new atomic components as extensions of the basic ECD++ via the C++ development plug-in of Eclipse. The IDE provides the binary executable for the appropriate target through cross-compilation mechanisms and provides means to download all the required files to the target platform to run simulations or an emulation of the model. The emulation was done by using real input and output capabilities through an IEEE-1284 compliant port, which have been added to the original ECD++ simulator in order to allow the test of hardware-in-the-loop techniques.

As an example of the use of the new functionalities of ECD++ a simulation and testing environment for an autonomous robotic system was developed. This environment applies modeling and simulation methodologies and the new testing methods to test a hardware-in-the-loop robotic system. In particular, the work on an autonomous vehicle simulation

allows us to proof the concept of having *real* and virtual simulations of the developments done with ECD++. For instance, when developing a robotic system that includes electric DC motors, a real motor can be hooked up to the target to see if the real hardware performs as simulated on the computer.

The other important contribution of the thesis comprises a benchmarking tool for comparison between different implementations of the DEVS formalism and as a tool to compare the performance of CD++ from different versions of itself. DEVStone was developed to measure the performance of simulations running in a tool that makes use of the Parallel extension of the DEVS formalism. The work done in this thesis first adapted the DEVStone benchmark to the standalone implementation of CD++ [17] and then extended the tool with a new tool that generates models that are more complex. In addition to the implementation of DEVStone adapted to the standalone CD++ version, we also tackled the problem of the performance of our tool compared with a different implementation of the DEVS formalism named ADEVS (A Discrete Event System) [18]. The main advantage that CD++ provides is flexibility, by separating the development of the simulator core and the models that use the simulation engine; whereas ADEVS provides a single portable library that embeds DEVS functionality in programs developed with C++. Both implementations of DEVS have been developed and used in different academic environments.

We used the synthetic benchmark to analyze the performance of different models in CD++ and ADEVS, which allowed us to show the performance variations of the both implementations. Moreover, these results permitted us to characterize the execution time of a standard DEVS simulator. The benchmark can be used to determine which directions and decisions should be taken when updating or improving either tool's simulation techniques.

Furthermore, DEVStone can be used to aid the measurement and improvement of other existing simulation tools.

1.3. Thesis Organization

The Thesis is organized as follows: Chapter 2 provides theoretical background of the Thesis discussing the DEVS formalism and the implementations of the formalism in CD++ and ADEVS, as well as basic definitions of embedded systems and the current approaches in the usage of development tools for embedded systems, simulators and simulators for embedded systems. Chapter 3 discusses the original DEVStone, the different models and the adaptation to standalone CD++ and ADEVS plus the new model developed for DEVStone. Chapter 4 presents the results of the DEVStone benchmark, from CD++ and ADEVS, and shows a procedure on how to use the results of the benchmark to improve the CD++ simulator. Chapter 5 focuses on the design of the Integrated Development Environment for ECD++ with a brief introduction to Eclipse and the CD++ Builder toolkit. In Chapter 6 RoboCart is presented, which is the embedded hardware-in-the-loop design test case presented; it was developed in its entirety using the new ECD++ IDE. The example uses a LEGO Robotic Cart connected to an Embedded PC running an ECD++ version capable of simulating and emulating models in real mode. Chapter 7 concludes this Thesis report and discusses about future research directions.

2. DEVS

DEVS is a mathematical formalism that is used as the basis of a M&S framework. One of the many advantages of DEVS is that it allows the construction of hierarchical and modular models, coupling of components, and even support for continuous-like discrete event model simulation by time approximation.

Given the natural hierarchical platform of DEVS, it allows the coupling of existing models modularly in order to build bigger and more complex systems. Because the formalism is closed under coupling, a coupled model can be treated as a basic DEVS component. The modular specifications of DEVS view every model as blocks with input and output ports through which all of the interactions between the exterior, and the internal and middle blocks –if any– occur.

A DEVS *Atomic Model* is formally described as follows:

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

Where:

$\boldsymbol{X} = \{(p,v)/p \in \textit{Input Ports}, v \in X_p\}$ set of input ports and acceptable values

$\boldsymbol{Y} = \{(p,v)/p \in \textit{Output Ports}, v \in Y_p\}$ set of output ports and acceptable values

$\boldsymbol{S}$: set of sequential states

$\delta_{int}: S \rightarrow S'$ internal state transition function

$\delta_{ext}: Q \times X \rightarrow S'$ external state transition function, where:

$Q = \{(s,e)/s \in S, 0 \leq e \leq t_a\}$

e = total time elapsed since the last state transition

$\lambda: S \rightarrow Y$ output function

$t_a: S \rightarrow [0, \infty)$ time advance function

At any time, the system is in some state defined in the set ***S***. In the absence of external events, the system will stay in the state for the time specified by t_a, which can be any real value between [0,∞). When t_a is finite and is consumed, the system first outputs the value λ and then changes immediately to a new state from the pool of states in ***S***. If an external event X is received before the expiration time t_a, the new state of the system is determined by δ_{ext}, where e is the time elapsed since the last transition. In other words, the state of the model is driven by the internal transition function if no external events are present, if an external event is received before the determined timer finishes counting then the state of the model changes accordingly.

A DEVS coupled model, composed of several atomic or coupled sub-models, and is formally described as:

$$M = \langle X, Y, D, \{M_i\}, \{I_i\}, EIC \mid EOC \rangle$$

Where:

$X = \{(p,v)/p \in Input\ Ports, v \in X_p\}$ set of input ports and acceptable values

$Y = \{(p,v)/p \in Output\ Ports, v \in Y_p\}$ set of output ports and acceptable values

D = set of component names; the following requirements are imposed on the components, which must also be DEVS models:

For each d ∈ D, $M_d = \langle X_d, Y_d, S_d, \delta_{int}, \delta_{ext}, \lambda, t_a \rangle$ is a DEVS with

$X = \{(p,v)/p \in Input\ Ports_d, v \in X_p\}$, and

$Y = \{(p,v)/p \in Output\ Ports_d, v \in Y_p\}$

Component couplings are subject to the following requirements:

External Input Couplings (EIC) connect external inputs to component outputs,

$EIC \subseteq \{(N,ip_N),(d,ip_d)) \mid ip_N \in Input\ Ports, d \in D, ip_d \in Input\ Ports_d\}$

External Output Couplings (EOC) connect component outputs to external outputs,

$EOC \subseteq \{(d,op_d),(d,op_N)) \mid op_N \in Output\ Ports, d \in D, ip_d \in Output\ Ports_d\}$

Internal Couplings connect component outputs to component inputs,

$IC \subseteq \{(a,ip_a),(b,ip_b)) \mid a, b \in D, op_a \in \textit{Output Ports}_a, ip_b \in \textit{Input Ports}_b\}$

Select: $2^D - \{\} \rightarrow D$ is the tie breaking function for imminent components.

X is the set of input events; Y is the set of output events; D is an index for the components of the coupled model, and $\forall\ i \in D$, M_i is a basic DEVS (i.e., an atomic or coupled model), I_i is the set of influences of model i (i.e., models that can be influenced by outputs of model i), and $\forall\ j \in I_i$, is the i to j translation function. Coupled models are defined as a set of basic components (atomic or coupled) interconnected through the models' interfaces. The coupling specification consisting of the external input coupling (EIC) which connects the input ports of the coupled to one or more of the input ports of the components. The external output coupling (EOC) which connects the output ports of the components to one or more of the output ports of the coupled model; and the internal coupling (IC) which connects output ports of components to input ports of other components. The influences of a model define to which model outputs must be sent. The translation function converts the outputs of a model into inputs for other models. This function defines that the outputs of the model M_i are connected to inputs in the model M_j, where j is an element of I_i.

The DEVS scene has been very active among several academic institutions, and many of them have come up with different implementations of the DEVS formalism. A non-up-to-date list includes the following implementations of DEVS-based simulators:

- ◊ **ADEVS** [18] provides a C++ library based on DEVS. Users can use the classes in the library to build their own models.
- ◊ **CD++** [17] is a modeling and simulation tool implementing DEVS and Cell-DEVS theory, which supports stand-alone, parallel [19] and embedded real-time simulation [20].

- ◊ **DEVS/HLA** [21] is based on the High Level Architecture (HLA) [22] and DEVS. It was used to demonstrate how an HLA-compliant DEVS environment could improve the performance of large-scale distributed modeling and simulation environments.
- ◊ **DEVSJAVA** [23] is a DEVS-based modeling and simulation environment written in Java that supports parallel execution. It provides classes for the users to implement their own DEVS models.
- ◊ **DEVS-Scheme** [24] is a knowledge-based environment for modeling and simulation based on the DEVS formalism, supporting real-time simulation and control.
- ◊ **DEVSim++** [25] is an object-oriented software to simulate DEVS models; which was implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- ◊ **GALATEA** [26] is a simulation platform that offers a language to model multi-agent systems using an object-oriented architecture. The tool describes a real system as interacting agents.
- ◊ **JDEVS** [27] is a DEVS modeling and simulation environment written in Java. It allows general purpose, component-based, object-oriented, visual simulation model development and execution.
- ◊ **PyDEVS** uses the ATOM3 tool [28] to construct DEVS models and to create the code to be executed. Models are represented as a state graph used to generate Python code and then interpreted by PyDEVS.
- ◊ **SimBeams** [29] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis and visualization using DEVS.

2.1. Embedded systems and simulators

Throughout time many definitions of embedded systems have arisen, the modern definition that clearly summarizes the main characteristics of such systems is given by [30]:

"*An embedded system is a special-purpose computer system designed to perform a dedicated function. Unlike a general-purpose computer, such as a personal computer, an embedded system performs one or a few pre-defined tasks, usually with very specific requirements, and often includes task-specific hardware and mechanical parts not usually found in a general-purpose computer. Since the system is dedicated to specific tasks, [it can be optimized], reducing the size and cost of the product. Embedded systems are often mass-produced, benefiting from economies of scale... In terms of complexity embedded systems run from simple, with a single microcontroller chip, to very complex with multiple units, peripherals and networks mounted inside a large chassis or enclosure.*"

Based on the general definition provided above, it is safe to conclude that embedded systems refer to systems that are connected to the real world through sensors and actuators, and perform dedicated tasks with varying levels of complexity. Historically embedded systems were mechanical or electronic devices, with very low complexity for the former and low computational power for the later, which had the advantage of providing a rather fast and exclusive response to all or some inputs to the system. In particular, the electronic version of this type of systems was completely analog and had their niche in process control and automation equipment. With the advent of higher computational power at lower prices [31], the use of such systems and their complexity has augmented considerably.

Some common characteristics of modern embedded systems include:

- They are often networked amongst themselves,
- They must interact with concurrent real-world entities,
- They may contain very large and complex software components,
- They may contain processing elements that are subject to the constraints of computation resources (such as memory, CPU, network speed), cost, size, etc.
- They more often than not rely on restricted energy availability,
- They may require an exact and timely output for a given input,
- Their development is done by higher power computational equipment and then downloaded to such systems.
- They may have one or multiple means for communication with similar or different types of systems.

Maybe the most important characteristic of any microprocessor-based embedded system regarding its software is the certainty of the system to respond appropriately and exclusively to inputs coming from its attached sensors. This last characterization applies to a huge variety of systems ranging from purely time-driven to purely event-driven systems.

For these systems, a systematic time handling and time modeling approach is usually not feasible to attain because of the multiple variations in the environments where these systems work. Since a systematic design is not always possible the validation and verification of embedded systems is accomplished through extensive testing, which includes heavily use of simulations in the early design phase [32]. The very nature of most embedded applications calls for stringent requirements for high reliability, which could be formulated by the intrinsic need for dependability and safety. However, and precisely because of the non-systematic approach of the design, the original design objectives are

usually compromised by non-ideal implementations that, sometimes, bear no resemblance with the original design techniques, i.e. the design is initially done with mathematical control models whereas the implementation uses UML-based JAVA or C++ tools.

The levels of reliability and safety often require fault-tolerant hardware and software, and make the testing of such systems of equal importance, or even of more importance, as the design of these systems. As a result, and given the impracticality of testing every possible scenario designers tend to simulate as many real worst-case-scenario tests as possible.

Embedded systems are often real-time systems, meaning that the time at which the system produces an output is finitely constrained, with the purpose of providing 'real-time' response for certain or some system's responsibilities. For this reason, the terms real-time systems and embedded systems are usually referred together as real-time embedded systems, because of the exclusive attendance of tasks by embedded systems, they are better suited to perform such tasks faster than general-purpose computing systems. The tight interaction between hardware and software that exists among many of these systems makes it difficult to separate completely one from the other, the software being heavily dependant on the hardware platform in which it will be executed. Nevertheless, for the purpose of this Thesis, we will refer simply to the software side of embedded systems without major concern in the limitations of the underlying hardware.

Most of the current research on embedded systems is focused on the operating system mainly because this is the *single* element that has to provide fast, predictable and concurrent services (such as fast response to interrupts and predictable scheduling algorithms to the programs running on top of it). These specialized operating systems are often stripped-down versions of traditional timesharing operating systems, which are made appropriate for the embedded domain [33]. An essential difference, due to the usually

unattended nature of embedded applications, is that for external events the related internal processes and outputs *must be* delivered, most of the time within a deadline. For the purpose of this research, we will refer to embedded systems that have to respond to external events in a real-time manner, since this the usual scenario where embedded systems are used.

When designing real-time embedded systems, the most common scheduling algorithms are Rate Monotonic scheduling [34], Earliest Deadline First scheduling [35], Minimum-Laxity-First scheduling [36] and Maximum-Urgency-First scheduling [37]. Included in the middle layer are computation models that are widely used in the design, analysis, and implementation of real-time embedded software. Formal computation models for embedded real-time systems have received growing attentions in the recent years. A formal model is an essential ingredient of a sound system-level design methodology because it makes it possible to capture the required functionality, verify the correctness of the functional specification and synthesize the specification tool-independently [38]. As timeliness is often an important feature in real-time embedded systems, computation models can be characterized into two categories: models not considering time such as finite state machine, Petri Nets, process algebra; and models considering time such as timed automata, timed Petri Nets, temporal logic. These computation models provide the basis to capture the behaviour and structure of a system under development. Those models considering time also capture the timeliness feature of the system. They support time modeling explicitly so are naturally fitted into the real-time domain.

While simulation methods help to analyze and design the systems under development, they face a common deficiency—that the simulation models are discarded as unusable by the implementation stage [39]. More often than not, the implementation techniques are not derived in any direct way from the simulation models. This *discontinuity* between the actual

implementation and the analysis, design, and modeling is a common deficiency of most design methods. It results in an inherent inconsistency among the different phases of design, implementation and test.

The simulation-based approach can be defined as a methodology that models a real system, and based on this abstract representation control models are constructed and tested through heavy use of computer simulations [40]. This approach can be used as a general tool for the design of a complete complex system or a specific tool, i.e. the design of an independent component of a complex system. During different development stages, different models of the same process are used depending on the purpose and design methodology used.

Ensuring consistency among different development phases it is an ongoing research topic in various areas of design. In software engineering, *traceability*, in the form of requirements traceability [41] or design-code traceability [42], has been advocated to ensure consistency among software blocks of subsequent phases of the development cycle. Boyd [43] shows how traceability can be achieved when designing reactive systems. In hardware/software co-design, Janka et al. [44] described a methodology that allows the specification stage and design stage to work together coherently when designing embedded real-time signal processing systems. These approaches use different methodologies for different stages. Design of real-time embedded systems can be improved by supporting consistent methodologies among all the design phases. For complex real-time embedded systems where multiple crews of engineers work on different aspects of the design, implementation and validation, it is very difficult to manage the software's complexity during development without the support of a continuous model.

The explanation given by Zeigler and Hu about *model continuity* indicates a methodology that keeps consistency among all development stages because *"the same control models*

that are designed and tested by simulation methods can be deployed to the real target system for execution". Because the control model remains unchanged from the design stage to implementation stage, no transformation or reconstruction is needed, more over the originally designed and simulated control algorithms can be deployed to operation seamlessly. This gives the simulation-based approach a decisive advantage among other methodologies, with it, designers can be confident that the final system in operation is the system that was designed and that the system will carry out the functions as tested by simulation.

The Thesis is based on different efforts closely related by applying simulation-based design. The conceptual approach presented supports the design of distributed systems via iterative refinement of a partially implemented design where some components exist as simulation models and others as operational subsystems. In [45] the authors present a simulation and control tool that provides the capability to model, as well as to control, real-world systems. Part of this research focuses in the development of a continuous model Integrated Design Environment framework, based on Eclipse, and the required adaptation of real-world control capabilities for the current version of the CD++ DEVS tool

Other methods for real-time software system development have focused on exploring the modeling capabilities for real-time embedded systems. For example, the unified modeling language for real time (UML-RT) [46] extends UML models to address special aspects of designing real-time systems. Kim [47] uses the time-triggered message-triggered object (TMO) model to capture the timeliness and concurrency properties of real-time.

The simulators used in this thesis make use in one form or another the DEVS modeling formalism as the basis for the construction of models and the consequent simulators. Both simulators run under Linux and are optimized, to certain degree, to make the best use of

system resources when running. Both simulators differ in the implementation although both are written in C++. A short description of each simulator, and their different versions is given next.

2.2. *ADEVS*

ADEVS (A Discrete EVent System) simulator was developed by Jim Nutaro of the University of Arizona. ADEVS is a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic DEVS formalisms. The models are constructed based on a template of classes in C++ and then compiled and linked to the library to produce the simulation executable. The latest stable version of the ADEVS template, as of this writing, is 2.0.5.1.

Every atomic or coupled model in ADEVS is comprised of two files:

- A library file (.h); where the name of the model, input and output ports and local variables are defined for the particular atomic model.
- A source code file (.cpp); where the actual model is built based on a template, common elements of the class include: constructor, internal transition function, external transition function, time advance function, output function, and destructor.

Once the model is written as a C++ file the **main()** function needs to be created in a new file. When compiling and linking all the code, the resulting file is an executable that has the simulator embedded in the model file. As a result, the model binary file is generally of larger size than the counterpart in CD++. In addition, the compilation time for medium to large models, which is negligible for CD++ compared to the simulation time, is not

negligible in ADEVS for some models might take more time to compile than to execute. . In the executable file, where the macro-model was defined and the simulator was created for the model, there is something to notice: destructors are inside the model itself and not in the main function, which the simulator is created, and the 'destruction' process is started from the inside out as usual but with one quirk, the simulator relinquishes its resources after the model has done it first.

Additionally, due to the self-contained characteristic of ADEVS, the compilation can be done with any ISO 14882 compliant C++ compiler without major problems. However, ADEVS was developed in Linux for UNIX like environments and the reality is that some caution has to be taken even when compiling with different versions of the GNU C++ Compiler.

2.3. CD++

DEVS is a formal Modeling and Simulation framework based on generic dynamic systems concepts. One of the main advantages of DEVS in respect to some other techniques is that it allows the implementation of the simulation core engine and the incumbent model to be completely separated from each other. In particular the CD++ [48] implementation, takes advantage of this characteristic because by doing it the verification and validation of both, simulator and model (the simulator is essentially another DEVS model), can be done independently. As a result, CD ++ permits reuse of prior built models, therefore if there could be a fairly big library of elementary atomic models it is possible to say that bigger and more complex models can be built from the existing ones and this coupled models in turn can be used as 'atomic' models for even more complex model constructions with as many interconnections among coupled and atomic models as the model requires.

The CD++ tool and the Eclipse-based front end CD++ Builder are ongoing research projects that implement the DEVS formalism for discrete event simulations. On the other hand, the CD++ tool and the Embedded CD++ version share common design roots, with the exception that Embedded CD++ is an optimized version of CD++ designed for reduced footprint in Embedded Systems. To avoid repetition, we will proceed to explain the Embedded CD++ tool, emphasising that the same functional description holds true for CD++.

2.4. Embedded CD++

As stated Embedded CD++ [49] is a stripped-down version of the more general CD++ tool, both tools are built as a hierarchy of classes in C++, where each class corresponds to DEVS defined entities. The two main abstract classes are the *Model* and the *Processor*. The former used to represent the behaviour of the atomic and coupled models, and the later deals with the simulation mechanisms. Figure 1 shows a simplified structure of both.

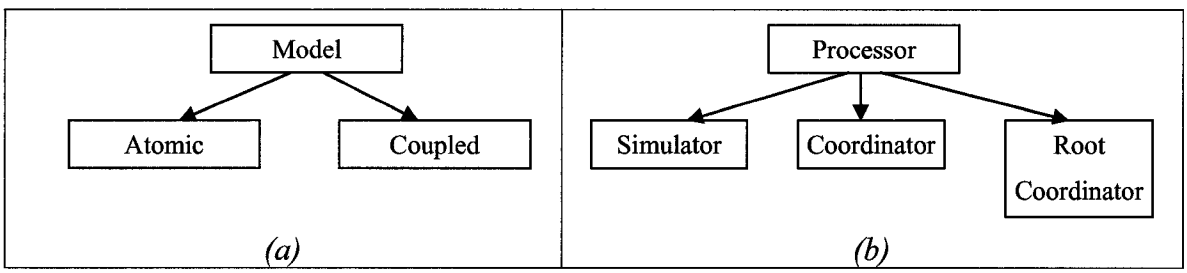

Figure 1: CD++ (a) Model hierarchy, (b) Processor hierarchy

The *Atomic* class implements the behaviour of atomic components and the *Coupled* class implements the equivalent mechanics for coupled models.

A *Simulator* object manages an associated atomic block, handling the execution of δ_{int}, δ_{ext}, δ_{con}, and $\lambda_{(s)}$ functions. A *Coordinator* block manages an associated coupled object. Only one *Root Coordinator* exists in a simulation and is manages the global aspects of the simulation. It is connected to the higher-level component(s) of the model, the *Root Coordinator* also controls the global time and starts and stops the simulation process. In addition, it is the one in charge of receiving the outputs of the model.

The simulation is accomplished by the exchange of messages among the components, for example, processors exchange messages to advance the execution of the model. Each message contains information to identify the sender and the receiver. A time-stamp for the message and an associated value are also included in the packet. For our purpose, it suffices to say that two categories of messages exist and each category contains several types of inter-component messages and administrative messages.

All versions of CD++ provide a unique specification language that allows describing coupling of models, initial values and external input events (in the real-time implementations of CD++, the expected output port and the expected completion time for an external transition can also be defined). For Embedded CD++ the complete development process was done in an entirely text-mode environment under Linux; whereas for the particular case of the standalone version of CD++ Builder for Windows or Linux, *Atomic* models are developed in an Eclipse-based environment in the C++ language; in this toolkit the combined use of an IDE for the development of C++ code provides greater flexibility. When adding new atomic models, each of them must inherit from the *Atomic* class in order to extend their basic behaviour. The *Atomic* class defines different methods for the initial function, internal and external functions and output function.

Until now, the only way to compare the performance among different versions of the CD++ simulator is by creating different sample models. Nevertheless, a benchmarking tool could

help better distinguish the advantages between one version of the simulator and another, while permitting comparisons with other DEVS simulators. The next section presents a short introduction to benchmarks and in particular a common benchmarking technique.

2.5. Benchmarking for Simulations and Simulators

As computer systems evolve is becoming more difficult to analyze the global performance of a system. Computer components on the other hand have developed separately their own benchmarking and performance measures. Standards and vendor specific synthetic benchmarks exist for processors, hard disk drives, random access memory (RAM), external peripheral buses, protocols, and operating systems. However, and due to the general-purpose nature of computers it is not possible to provide a general benchmark for a wide range of applications.

When it comes to test computer systems in particular, *application benchmarks* are preferred to synthetic benchmarks, because they reflect a real performance of the systems under test by running real-world applications [50]. In cases where is unfeasible to run a batch of real-world applications *synthetic benchmarks* come on handy given that, they can provide with an *approximate* degree of certainty the performance of a computer system under test by executing *artificially* designed workloads that resemble the real-world application's workload.

Very few efforts have been derived on the performance analysis of simulators, and in particular of discrete event simulators. Most commercial simulators are compared against each other based on the amount of features or suitability-to-task [51] rather than on a systematic way. In the case of academic efforts, the comparison between different

implementations of same algorithms varies from well-grounded scientifically based benchmarking to non-existent. This because analyzing simulators can be an extremely complex task; end-users can create a wide variety of models with different structures, levels of complexity and mixed degrees of interaction between models. Most studies of simulation techniques are focused on very specific tools and algorithms. In particular, existing performance studies devoted to DEVS-based simulators cover almost exclusively parallel and distributed implementations. For instance in [52], the performance measures of Cell-DEVS models in a parallel environment; in [53], a watershed model is used to demonstrate performance improvements in parallel and distributed architectures; in [54], the performance of DEVSCluster is compared with the performance of D-DEVSSim++; for the comparison of DEVS-based simulators against continuous-time type of simulators Zeigler [55] demonstrates that DEVS is more efficient when simulating natural and artificial systems. In the particular case of the CD++ implementation of DEVS, an interesting approach was introduced as DEVStone.

DEVStone is a synthetic benchmark that provides thorough analysis for the execution of models with different characteristics; in addition, it provides a common metric to compare results among different DEVS-based simulators. The accuracy of DEVStone results is based on a large pool of models that when combined can provide a robust test set. DEVStone is able to generate different models that vary in size, complexity and behaviour that have the same functionally of different kinds of real world applications. Based on predefined synthetic model it is possible to analyze the efficiency of a simulation engine, may it be a new version of CD++ or a different DEVS-based simulator, with relation to the characteristic of a category of interest.

DEVStone allows the developer to have control over the key factors of performance metrics in a simulator: the size of the model and the workload carried out in the transition functions. DEVStone produces models require the following parameters as input:

- Type: defines the different internal structure and interconnection schemes between the components.
- Depth: the number if coupled components or *levels* in the modeling hierarchy.
- Width: the number of *Atomic components* in each coupled component or level.
- Internal transition time: the execution time spent by internal transition functions, measured in Dhrystones per second.
- External transition time: the execution time spent by external transition functions, measured in Dhrystones per second.

With the flexibility provided by the benchmark, the original DEVStone showed how it can be used to test and optimize better algorithms or improved features of CD++. By using DEVStone to generate a set of small and large models with different parameters and running simulations of these models with the normal version and a modified version of the simulator, it was demonstrated that the creation of intermediate coordinators/simulators and the passing of messages among them created an excessive overhead in the Parallel implementation of CD++. To reduce such overhead, improve resource utilization and, in general, optimize the performance of the tool, flattened coordinator and simulator were used in the modified version of the Parallel CD++.

On the other hand, another step towards the design of an integrated, self-contained development tool is the development of an interface with the designer, which would include a mechanism for activating the different components of the simulation engine, including the Benchmarking tool. An Integrated Development Environment (IDE) would help to display

all relevant information on the screen as soon as is it produced by the system. In the case of the simulation of an embedded system, this interface is of the utmost importance, since it might be the only way to analyze the intermediate and output states of the system being analyzed. The development of such interface for the embedded version of CD++ is discussed in the next chapter.

3. Embedded CD++ Builder Integrated Development Environment in ECLIPSE

Working on ECD++ requires writing C++ code in a text-based Linux environment with open source tools. In order to improve the development and simulation experience, CD++ provides a IDE for the simulator core, which was developed for the CD++ Standalone version; the IDE plus the simulator was called CD++ Builder [REF1], and it was built on the Eclipse Environment [56] as a plug-in.

In the case of Embedded CD++ (ECD++) an IDE is necessary, because ECD++ will most likely be running on embedded platforms with minimum, or even none, output peripherals, therefore the information required during development is rather limited for the developer from the intended platform. We have extended the IDE provided by CD++ Builder, adding Embedded CD++ functionality. The concept behind this is to permit the developer to port seamlessly already written code (code reuse) from CD++ Builder to Embedded CD++ without worrying about compatibility problems.

In order to have this environment integrated with the original CD++ Builder tool some basic requirements and design considerations need to be fulfilled:

- The IDE for ECD++ should permit code reuse from the original CD++ Standalone version, ideally sharing all the possible resources that the development environment has to offer from the later. Ideally, it would be integrated within the actual environment.
- Since ECD++ will be deployed in a different platform other than the one where it is being developed, cross-compilation will be necessary.

- Means of communication to the Target platform have to be part of the tool, in order to download the executable binary file, running the executable and for remote debugging and maintenance if required.
- In order to make the tool easy to work with, it should remember important 'preferences', i.e. last IP Address used if the connection is established through a Local Area Network, or other important information that remains constant throughout the development process.

The graph in Figure 2 summarizes the additional tools needed to achieve the functionality that the design objectives state. From the CD++ Builder plug-in, five new processes need to be spawned, each one parallel to the others but also following a certain order among themselves, for instance the project needs to be edited first in order to be compiled and generate an executable file. Only when this file is obtained it can be deployed to the embedded target, and only when this file is present in the target it can be run remotely thorough a remote shell connection or remotely via a remote command. However, each process is separated from the other to give the user complete control over the development, for instance, the project can be compiled but not deployed and a previous version of such project can be executed for testing purposes.

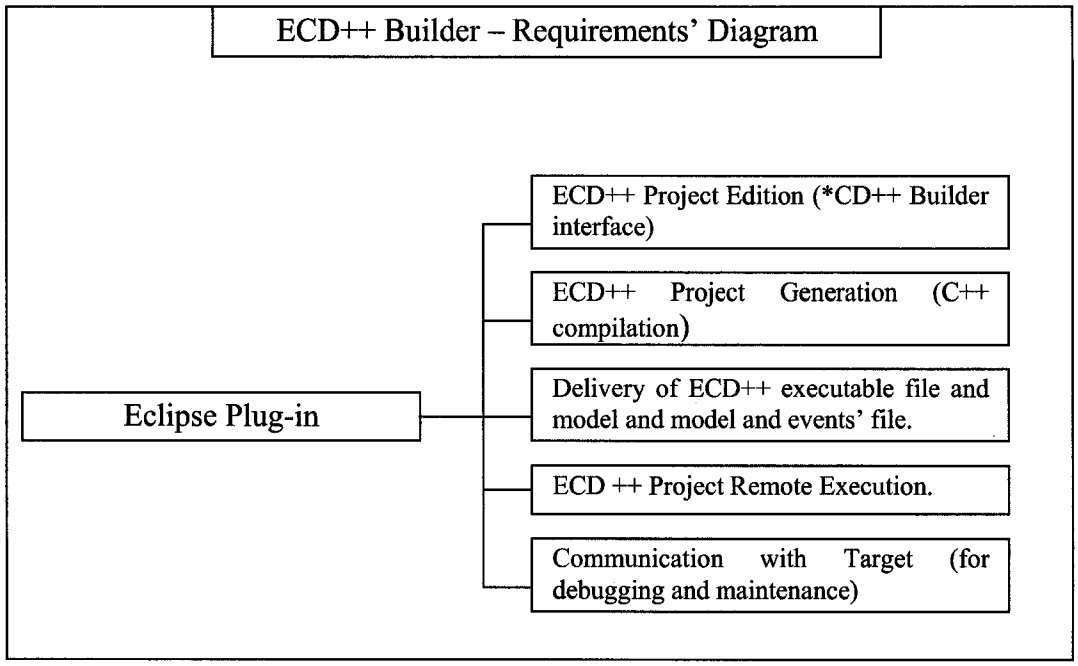

Figure 2: ECD++ Builder IDE as Eclipse plug-in – requirements

Based on these requirements and the actual implementation of Embedded CD++ and CD++ Builder, there are some limitations of the design, which are:

- To allow flexibility on the Target, the cross-compiling implementation is setup by modifying three files, in other words before using the tool a working cross-compiler must be set up, *then* the path to compiler needs to be updated to three configuration files that come with the plug-in.
- The link to the Target relies on services provided by different kernel services and additional software, therefore the Eclipse IDE provides a 'hassle-free' experience for the model developer, *after* the initial required software components have been installed; some of the services that IDE makes use for the Embedded CD++ Builder version are: telnet, ssh, java, bash, X-server, etc.

- Most of the communication's functionality used in the development is only available to the `root` user by default; therefore the IDE assumes that any developer using the tool has `root` access, or equivalent, to the system.

3.1. Embedded CD++ IDE - Development

Four new features need to be added to the tool:

- **Compile2Target** - Allows the compilation of the software with the cross-compiler, with a similar methodology as the one used for the Standalone version, with some modifications to adapt the automated process to the ECD++ tool.

- **Download2Target** - A new feature inside the plug-in that allows the downloading of the binary file to the Target platform by establishing a Network File System (NFS) mount between Host and Target. Whenever NFS 'mount' is set up the Host downloads up to three files: the ECD++ simulation binary, the model file and the external event file if any or both are selected, when the copying of the files is finalized the NFS folder is 'unmounted'.

- **Run Simulation on Target** – Allows the execution of the simulation remotely from the Host machine, with user selectable parameters, redirecting the display output of the Target machine to a non-interactive Console window in Eclipse.

- **Telnet2Target** – The last feature offers a way to establish a remote connection with the Target, which can be used to execute the simulation, to debug such simulation

remotely by using standard Linux remote debugging tools or for maintenance purposes, i.e. to configure network parameters on the Target.

In the end of the development process the IDE main window looks like the one shown in Figure 3, where the buttons providing the new features are circled.

Figure 3: New CD++ Builder environment window – Embedded CD++ functionality

For the **Compile2Target** feature, keeping consistency with standalone CD++, the first question that the developer is asked is if it is necessary to have a verbose output or not. Once the user selects the preference, this is stored in a preference field and the user gets no more questions about on screen information of all the process. Next the feature checks the availability of new model files (models and libraries) within the project folder, if found they should be moved temporarily to the folder that has the required files to generate the simulator executable.

In Linux the compilation and linking is done automatically through a *makefile* script file, for the successful compilation of the new model components in the simulator this file needs to be created based on a template that includes the new components of the project. Once the script is created the make command needs to be executed with the new file as a parameter.

For accomplishing these tasks the **Compile2Target** makes intensive use of some Eclipse services as well as of JAVA components. For instance, all the windows are made using JAVA graphical services, plus the output of all the required process running in the background are redirected to a JAVA Console window where all the messages are available to the user. The search for project files is done through Eclipse by checking the project file and looking-up the list of files based on the ".cpp" and ".h" extensions. The JAVA threading capability built in Eclipse is used to execute shell programs in a different in the background, for instance the copying of the new project files, temporarily, to the internal directory where the compilation will be done. In addition, the threading is used to launch commands for moving the executable for the project into the original folder as well as temporary files. The generation of the *makefile* script is done using the JAVA file I/O functionality and, based on a template, copy the template character by character including the new files where necessary at the end of line. Finally the threading capability is called upon again and the *make* command executed in a separate thread with the generated script file as a parameter. The **Compile2Embedded** feature, like all the rest of the new features, is a self contained JAVA class that is called through the plug-in eXtensible Markup Language (XML) script. When one of the buttons is clicked on, the XML script launches the corresponding JAVA class, which contains all the required components to draw the required window and executes the task that was designed for. The class diagram of the new features of the CD++ Builder for Linux can be seen in Figure 4.

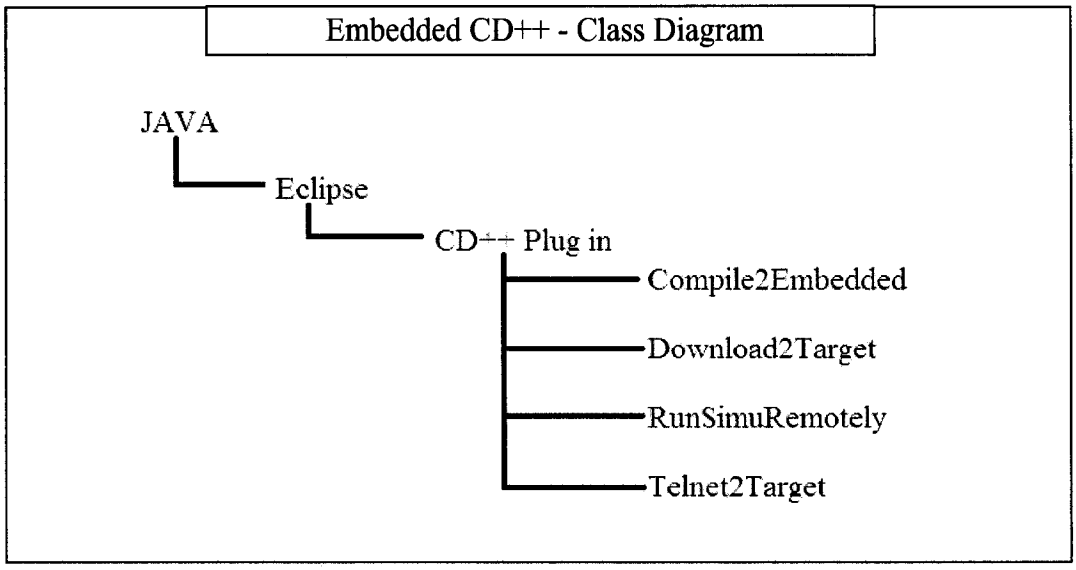

Figure 4: ECD++ Builder Compile2Target – Classes Diagram

The **Compile2Target** functionality needs supporting software running underneath Eclipse to perform all the required tasks. As stated, Eclipse gathers information on behalf of the new tool about the location of the files and the existence of such files. Once basic information is made available to the **Compile2Target** feature, it makes use of the JAVA threading capability that comes in Eclipse and initiates basic commands (i.e. cp, mv, make) to place project files in a temporary location. It also uses creates a text file based on a template using basic File I/O from JAVA. All the editing of the files is done using the IDE of the CD++ Builder plug-in. A diagram of the supporting software and its interaction with the **Compile2Target** feature is shown in Figure 5.

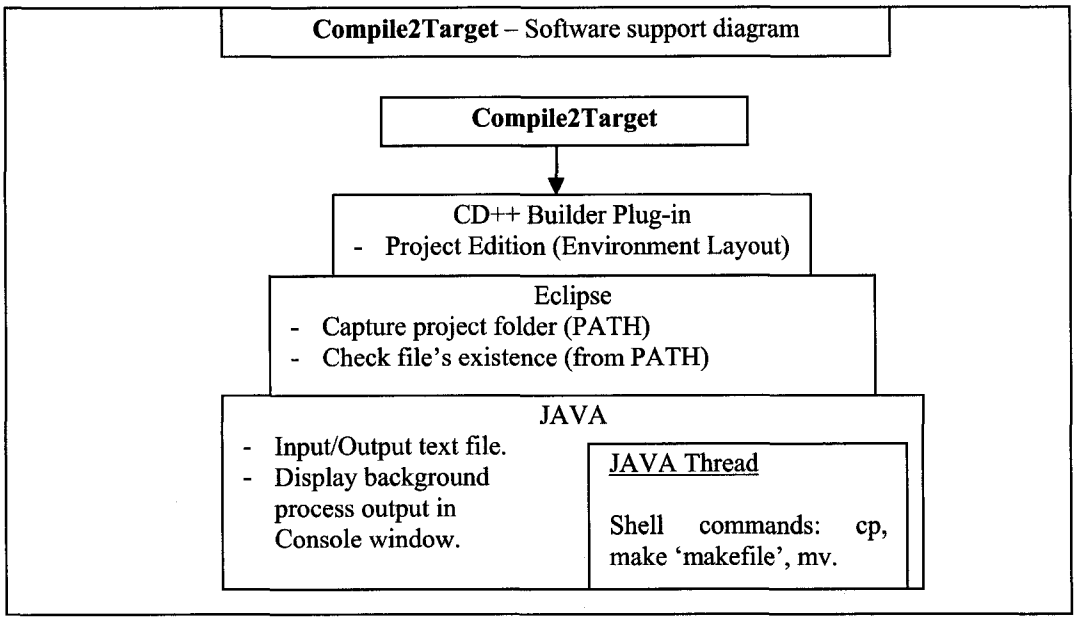

Figure 5: ECD++ Builder Compile2Target – software support diagram

Running the feature three different windows will be presented to the user, the first one asking for the verbosity option, which only occurs at the begging of the Eclipse session. Then a Console window redirects the output from the process running in different threads in the background, and a bar-graph progress window shows the advancing of the process. Finally the progress window is closed and the results and error messages if any are presented in the Console windows. The whole process is depicted in Figures 6 to 8.

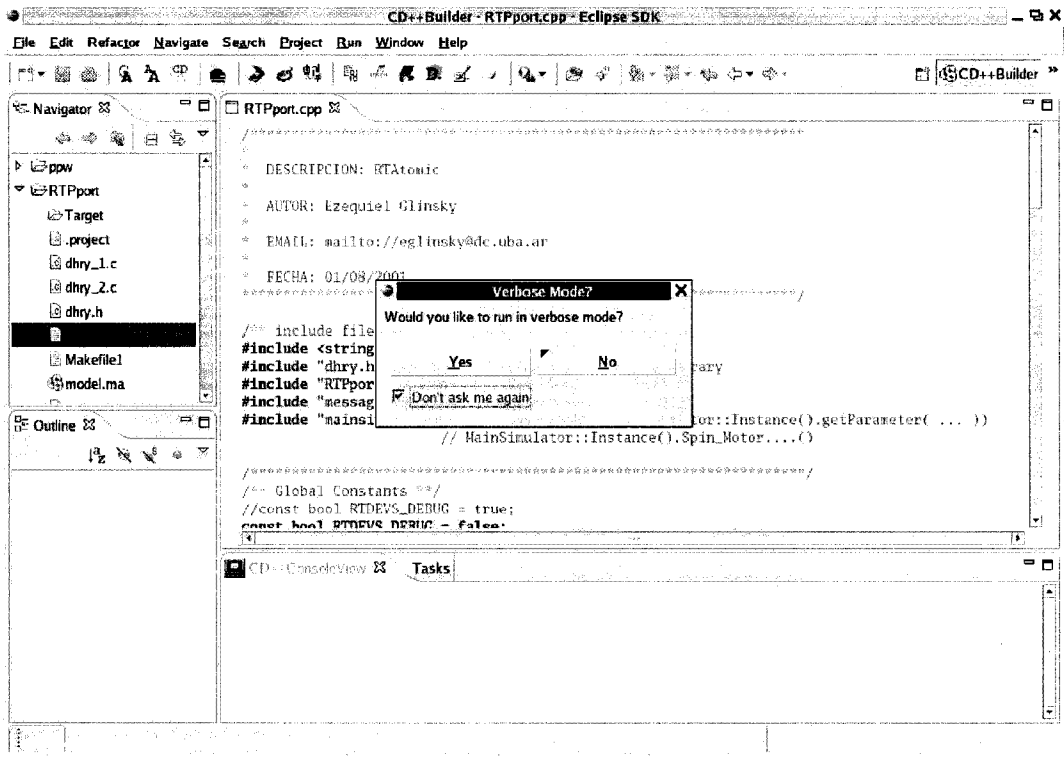

Figure 6: ECD++ Builder Compile2Target first window

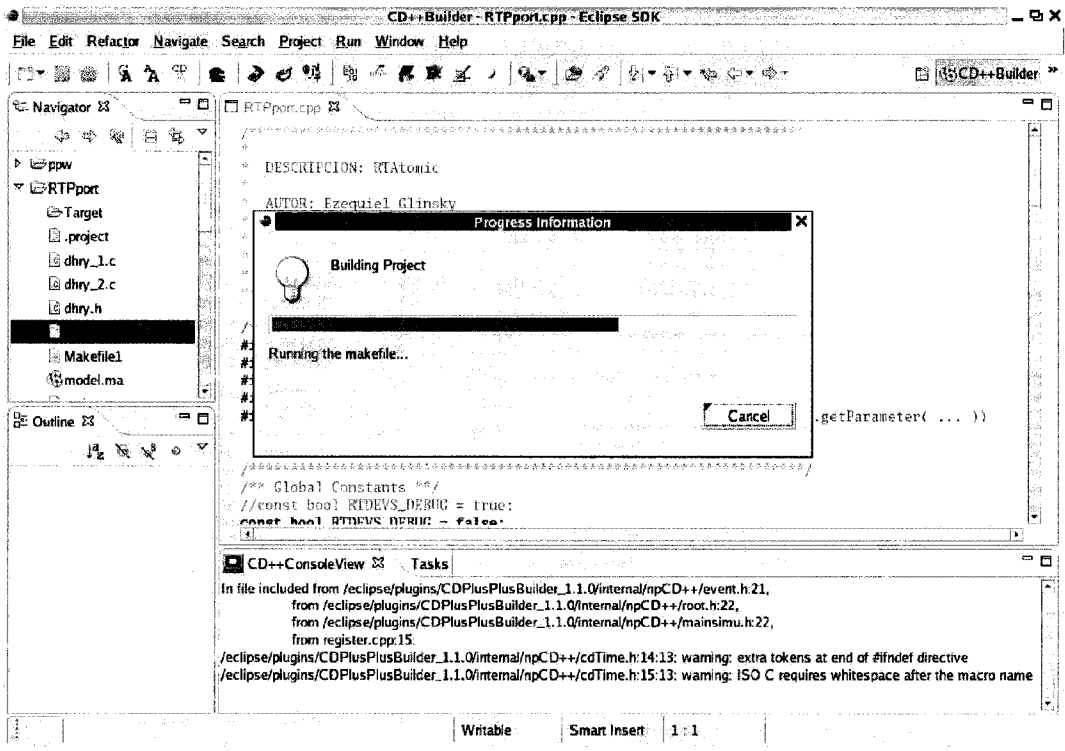

Figure 7: ECD++ Builder Compile2Target progress window

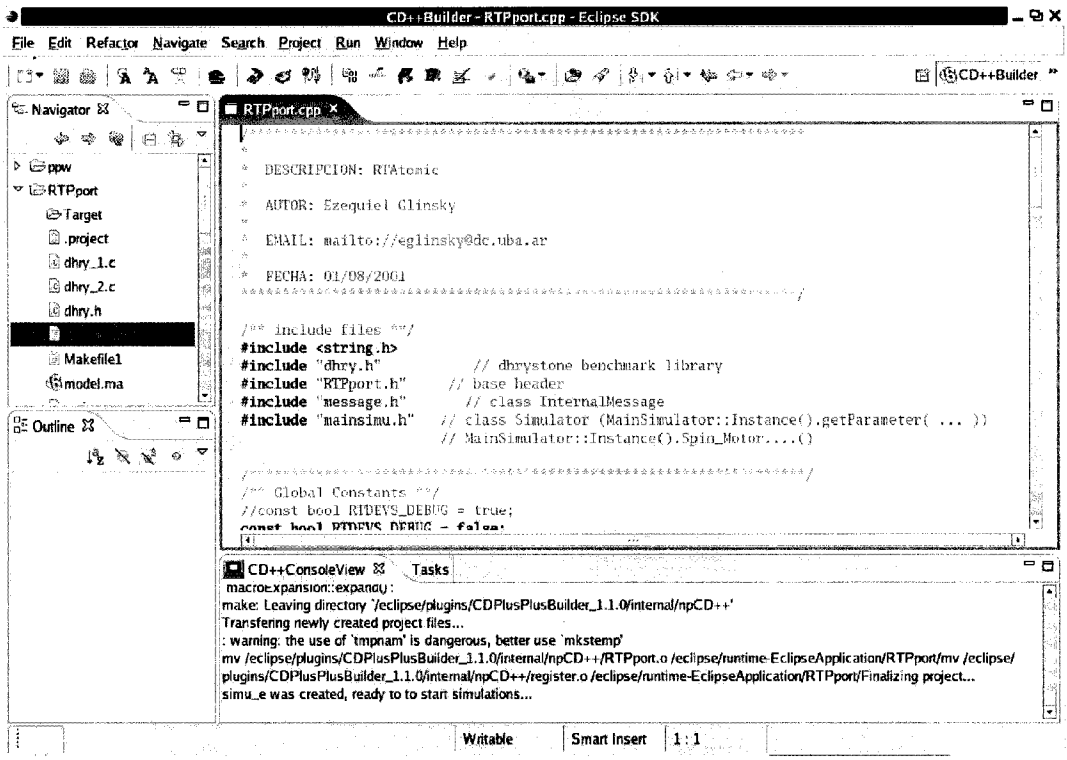

Figure 8: ECD++ Builder Compile2Target – IDE when **Compile2Target** finishes

The **Download2Target** functionality uses the NFS mount feature of Linux. Initially the plug-in looks for the existence of the binary file, if found it creates a folder within the project directory where the Target will be mounted. By calling the mount service on Linux much of the problems related to authentication and connection over the network are dealt by the mount utility and the operating system. One detail that was found during the development is that when executing the copy commands in different threads from Java, the execution is too fast for the scheme used, resulting in many threads trying to use the results of a command in a previous thread, i.e. the copy command at the same time that the mount NFS folder. As a result, the last task that is in charge of demounting the NFS folder is called for during the copying of the files, which throws an error of the network device being busy.

To overcome such a problem, a scheme that forces sequential execution of the threads was implemented. In general terms this scheme does not allow the threads calling for an execution of external commands to run in parallel, forcing the main thread to remain waiting in itself until the thread that was first created is terminated; i.e. the main thread waits for the NFS folder to be mounted and then waits for each file to be copied into the Target before copying the next file or before demounting the NFS resource. The IP Address field of the IDE is saved in the preferences file of the plug-in to save the developer the hassle of introducing the IP address every time he needs to download a new version of the embedded simulator. A remote folder field was created in case there is the need to have multiple versions of different simulators on the embedded device, if it has enough memory. Such configuration though, would require that the user create new access permissions for multiple folders on the Target before using them. The advanced options field provides flexibility and permits the use of virtual Targets of available, i.e. the virtual device is inside a folder or is a file that needs to be mounted with different parameters than an NFS mount.

The **Download2Target** feature design chart shown in Figure 9 depicts the relationship between the feature and the supporting software for the deployment of the executable file and the required files needed to run a simulation on the Target Platform. Initially the feature gathers information through a JAVA window with options fields where the user can explicitly type in the desired destination folder where the binary should be copied. Plus it gathers information about the location of the files that will be copied. To check the existence of the files introduced, an Eclipse service is summoned to check the in the project's path the files introduced. With this information the feature uses a JAVA background thread to 'mount' a networked folder on the Host. If this connection is established successfully then the feature initiates additional threads and copies the required files, one after the other, to the destination folder, as a last step the feature 'unmounts' the

NFS directory from the local folder tree. This feature does not use any functionality of the plug-in directly.

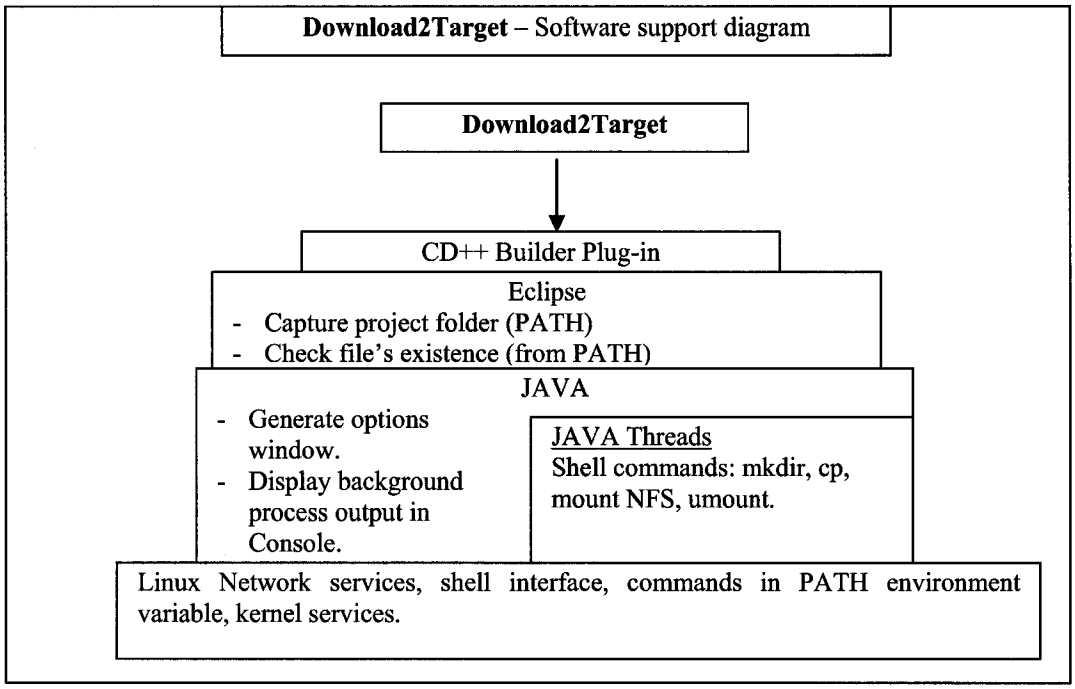

Figure 9: ECD++ Builder Download2Target – software support diagram

Figure 10 shows a snapshot of the options window that pops up every time that the **Download2Target** feature is executed, the target IP address and options fields can be seen. The selection checkboxes for the files are checked enabling the text boxes to accept inputs (the boxes are disabled when the selection checkboxes are unchecked). Something similar happens with the advanced options checkbox.

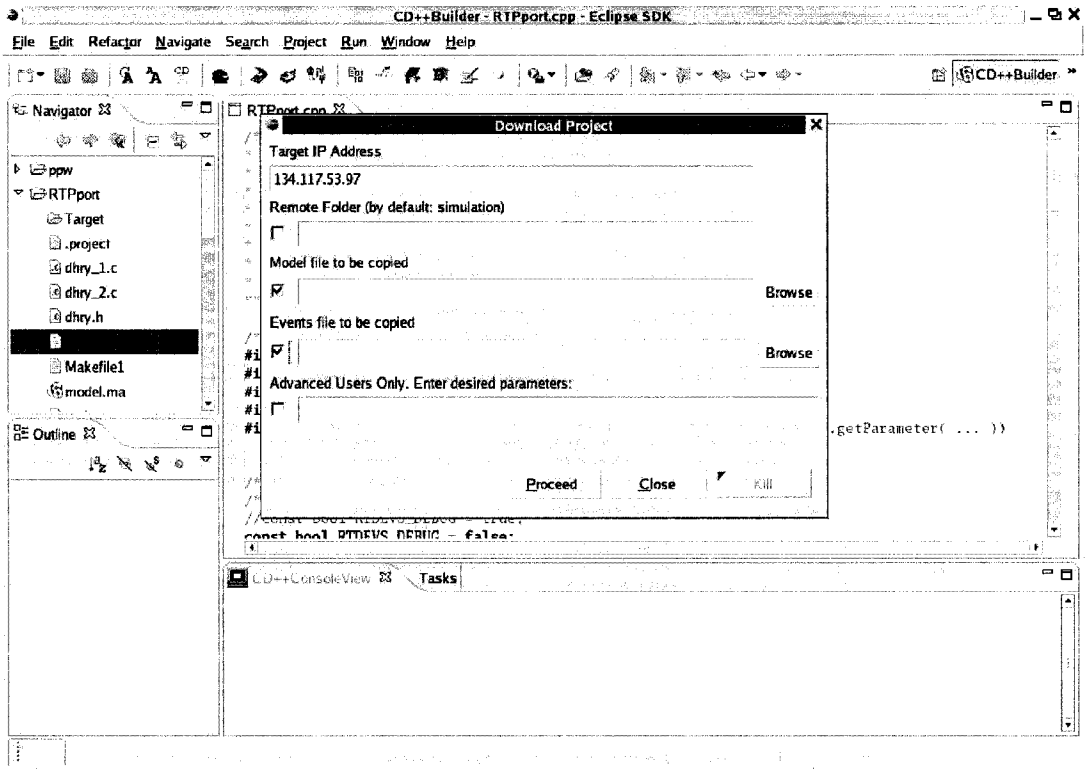

Figure 10: ECD++ Builder Download2Target window screenshot

Once the executable binary file for the appropriate Target plus the model and, if required, external events files have been downloaded there is still the need to **remotely run the simulation** on the Target platform. A solution to this problem is provided by a third new functionality that automatically generates a script file based on the options introduced by the user in a new ECD++ Builder window and then runs it and displays the remote output information in a non-interactive CD++ ConsoleView window. This functionality works for any model that is downloaded into the default `/simulation/` directory, if the executable file and the additional files were downloaded in different directories other than the default then the best alternative to run the simulation is to connect remotely to the Target and execute the simulation from a remote shell.

The method used to execute the remote command uses a *secure shell* call, therefore prior to the use of this functionality an ssh-keyword needs to be generated and shared by both

platforms [57]. This setup allows the execution of single commands from a *registered* host in the Target platform without the introduction of passwords or authentication.

The process for the remote execution of the simulation is rather simple. On pressing the remote execution button in Eclipse the class **RunSimuRemotely()** is summoned and, an option window pops up, allowing the user to introduce all the parameters desired for the simulation. After introducing all the parameters and the execute button is pressed in the options window, the feature mounts the Target default destination folder as a NFS device in the default Target folder (`/Target`) and a script file is created based on a template with the parameters introduced by the user. Using different sequential threads, this file is immediately made executable, copied to the NFS directory and the Target folder unmounted. Upon termination the feature runs a remote command execution (`ssh`) in a separate JAVA thread, redirecting the output of the process running in the thread to a Console Window.

Figure 11 shows the supporting software required to complete all the steps needed. The basic information is provided by Eclipse itself (preferences) then the feature executes different shell commands in a sequential manner, none of the steps can be executed in parallel, and the execution of one after the other is enforced. The commands executed only present a message if some thing goes wrong with the mounting of the folder or the creation of the file. On simulation the output of the simulation takes precedence and all the messages are redirected through JAVA to an existent Console Window in the Eclipse Environment. A picture of the parameters window for remote execution can be seen in Figure 12.

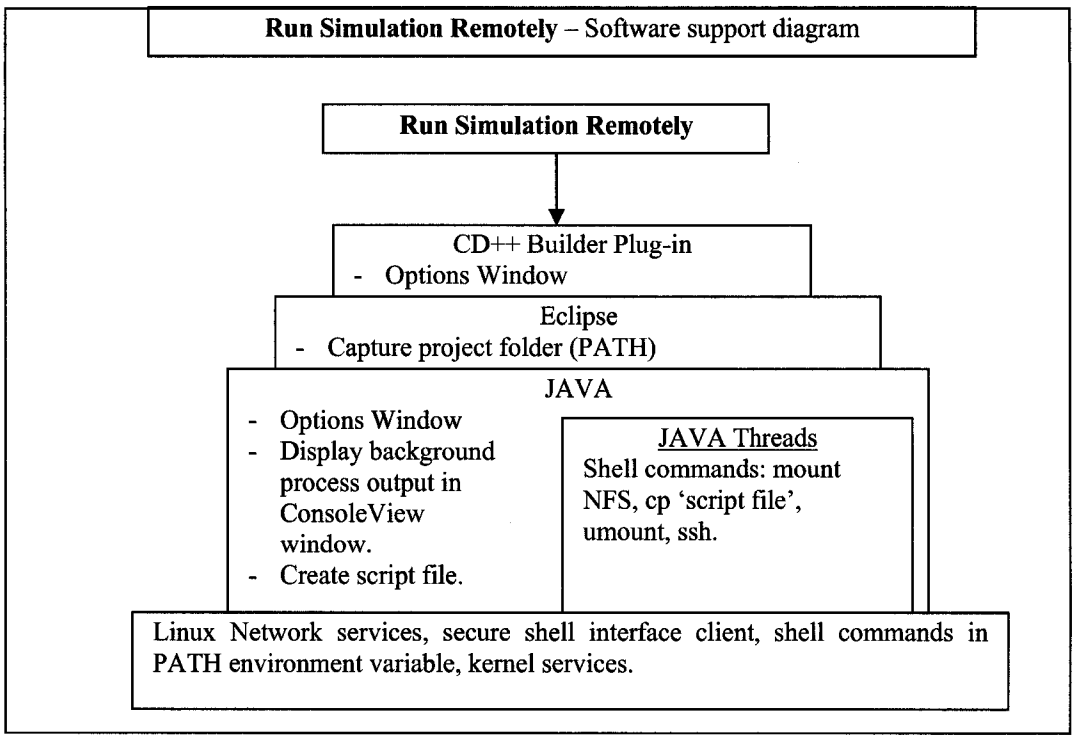

Figure 11: ECD++ Builder Run Simulation Remotely – software support diagram

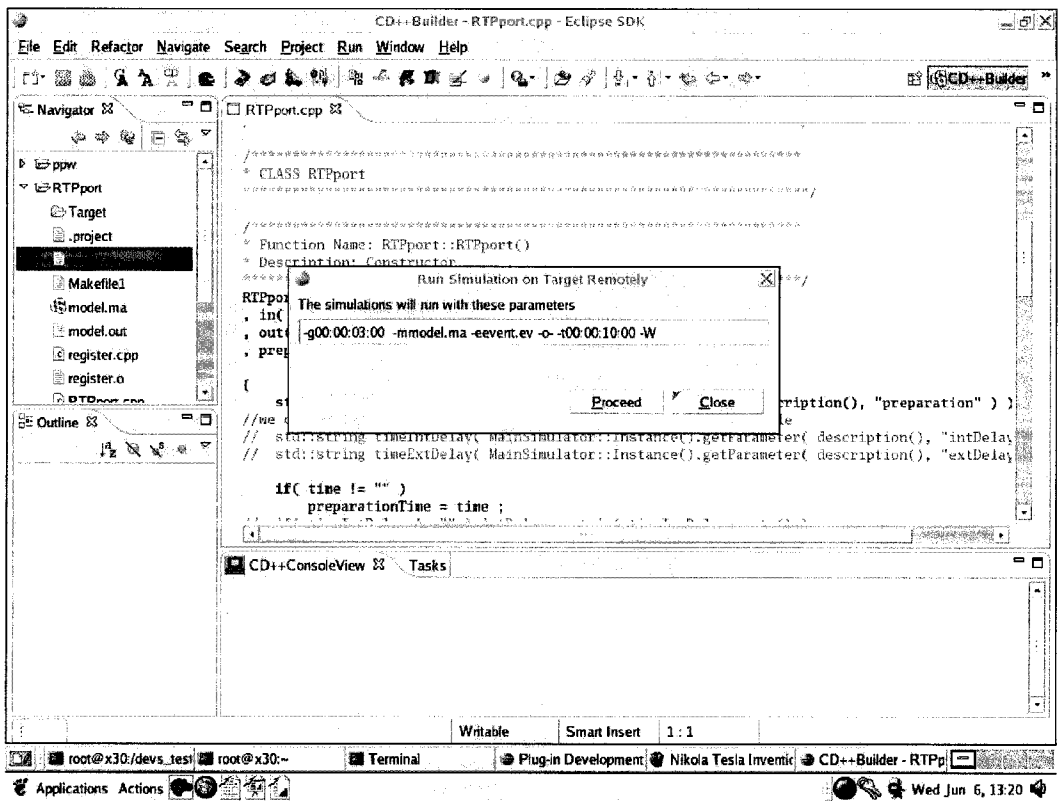

Figure 12: ECD++ Run Simulation Remotely – Parameter's Input Box

The last feature added to the CD++ IDE is called **Telnet2Embedded**. The primary function of this new functionality is to establish a communication channel form the Host to the Target to perform different tasks within the target device. The communication is established using Telnet mainly because the footprint of a Telnet server in the Target is small enough to be present in any type of embedded device; since these kinds of devices are known to have limited memory space; however, Telnet also increases the vulnerability of the system [58] providing less security in the authentication and communication than other types of network communication, in this project's the security of the Target system is not critical therefore it can be traded-off for smaller footprint.

Another advantage of Telnet is that is simpler to setup and modify the Telnet server side with minor effort from the developer, also there is the need to consider that there is a Telnet implementation for every kind of Linux distribution as well as Windows environments and other Embedded Real-Time operating systems. In order to work, this connection scheme requires the user to have prior access to the Target by other means other than the development tool to set up the Telnet server as well as the NFS server and permissions.

Figure 13 summarizes the required software support of the **Telnet2Target** feature. A window is presented to the user filled with information extracted from the plug-in preference's file. In the options window the last IP Address used to connect with the Target is displayed and the user is set to the root user by default. The feature executes the telnet command in a terminal window that is dependant of the X-server system. In Fedora there are two common options: GNOME and KDE. This is the reason why the options checkbox allows the user to change the Desktop Environment. The selection allows either one of them but not both, neither none. The selection of one will force the non-selection of the other. Once the information is accepted the main class is launch which consists of a single task: run the telnet command in the appropriate Terminal window for the Desktop

Environment. Figure 14 shows the options window that comes up when the **Telnet2Embedded** feature is summoned with the checkbox selecting Gnome as Desktop Environment.

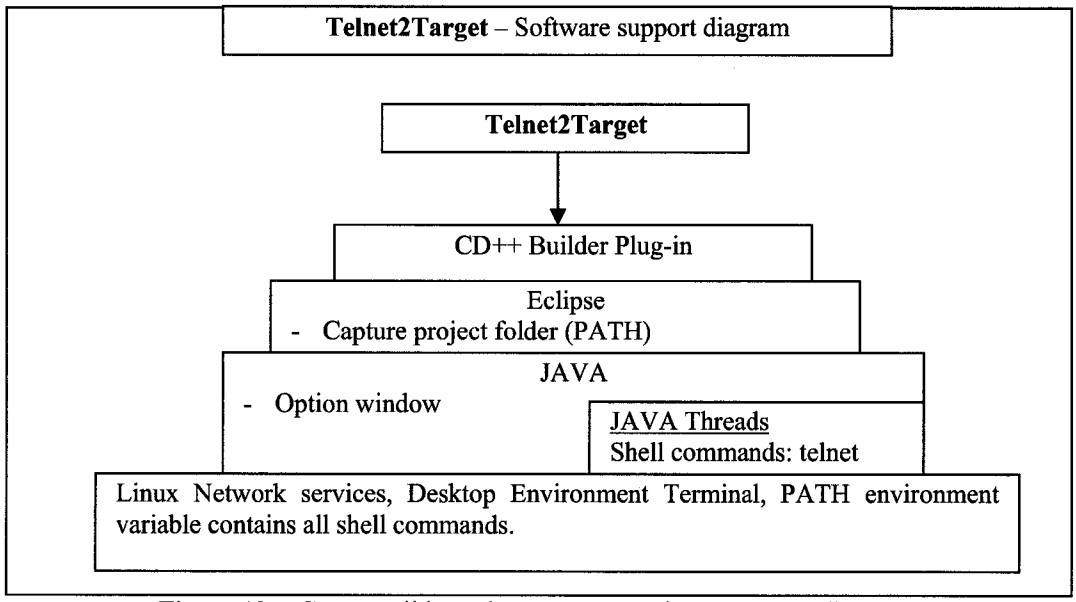

Figure 13: ECD++ Builder Telnet2Target – software support diagram

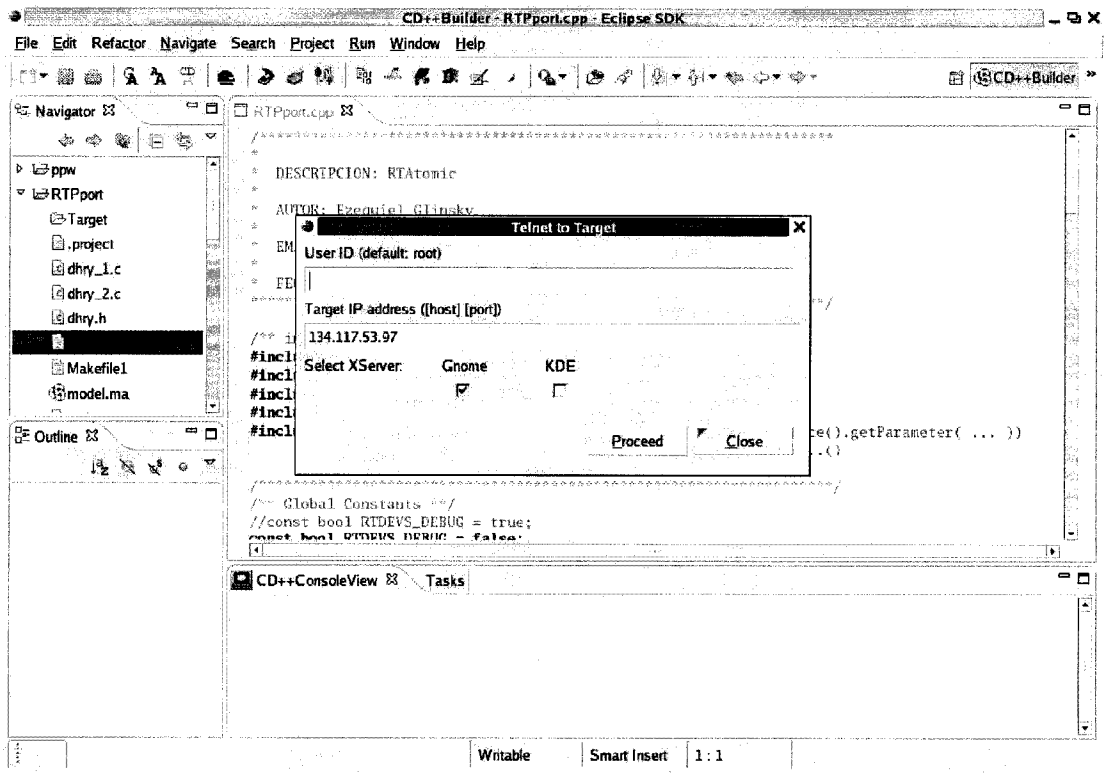

Figure 14: ECD++ Builder Telnet2Target window screenshot

Having built the interface for an embedded system, a good way of testing it is by developing a simulation with a medium level of complexity, for example the use of Hardware-in-the-Loop simulations require the modification of the simulator's core system, including timely response to inputs and the adequate handling of input and output data. Therefore, to show the capabilities of the new development tool and the flexibility of ECD++ when dealing with external events a common model was built, a semi-autonomous robotic cart which is capable to go around obstacles when they are found in its path, through the use of a touch sensor. The construction and testing of the model is given in the next section.

4 RoboCart

The fundamental use of CD++ is to be used as an academic tool for discrete event simulation learning. Though most simulators provide enough abstraction for the student to understand the principles, there is always the 'real factor' missing in this approach. At the same time, interfacing through a computer printer port has been, probably, the most used type of interfacing throughout the history of computing ranging from rather simple communication protocols via the standard parallel port or data intensive communication using enhanced version of the parallel port as described in the standard IEEE 1284. This port emits and receives TTL (transistor-transistor-logic) signals of 0 [V] and 5 [V]. The outputs of this port are latched by flip-flops, thus conserving the last value written to the port.

There has been a steady ongoing work in the Embedded CD++ front to run Real-Time simulations. Until now, the work done on Embedded CD++ provided us with a special option to run simulations in real time, using the computer's real-time wall-clock [51].

Using the new IDE for Embedded CD++ tour goal was to build a test system as quickly as possible with a medium level of complexity, which includes the development of Hardware-in-the-Loop simulation test system. By using automated common tasks during development, that had to be manually coded or typed in prior to the existence of the new ECD++ IDE tool, a project that would have a relative lead time of a couple of weeks was finished in 5 days since the conception of the model to the test and debug stage (including the development of the hardware and software components). The development took 2 days from the generation of the DEVS models to the implementation using CD++ templates, and the testing and debugging took the rest of the time, and was mostly due to debugging the

data coming from the Parallel port and modifying the appropriate program files of CD++. For the development of the HIL system, DEVS models of all the components used where created and coded accordingly the standard CD++ template. Because CD++ was also developed DEVS models, and is a DEVS model in itself, new DEVS models that dealt with the interaction of external events and the simulator's internal behaviour were also created.

In general, Hardware-In-the-Loop simulation is a dynamic test technique that simulates the I/O behaviour of a physical system that interfaces to a computer control system in real-time. It is dynamic because the values of stimulus signals generated by a simulator are a function of a computer's response from the previous cycle.

However, due to the slow nature of the peripherals compared to the processing speed of the computer's microprocessor there are some technical challenges at the time of the implementation. In the ECD++ case, when the simulator is running in real time mode, an event file is read at the beginning of the simulation and according to the information contained in the file the simulator engine asserts an external event when the real-time clock reaches the predefined time.

To be able to respond to external events is obvious that the simulator needs to be run in real time; otherwise, the simulation would evolve in a time scale too fast compared to the time scale of the real process making it impossible for the slower real events to catch up with the simulation. When running in real time, the ECD++ simulator requires the time-stamp of the external event and the time stamp of the expected finalization time of the simulator's response. The approach presented in this Thesis modifies the loading of events running the simulation individually for each received external event. This method is also DEVS-based and can be easily cast into a DEVS atomic model:

Parallel port read = $<X, Y, S, t_a, \delta_{int}, \delta_{ext}, \lambda>$

X: Parallel port external event: is a new event coming from the parallel port.

Y: output port: is a new external event with all the data required by the root coordinator to perform a complete run of the simulation, timestamp of the event, expected completion time, port name and value.

S: system states: forward external event from the parallel port; wait for next external event.

t_a: time advance function: the time advance is provided as real time count from the computer's real time clock.

δ_{int}: internal transition function: is the total time for the simulation to run, if infinite then this atomic component can only respond to external events.

δ_{ext}: external transition function: since there is no direct method to generate interrupts to the processor from the parallel port, the external transition function is implemented as constant polling and comparing the acquired value to the last value in memory, if these values are different then an external event is generated.

λ: output function: sends the value of the external event to the list of external events managed by the root coordinator along with the time stamp of the event, the expected finalization time, the input port and the value as a floating-point number.

Program Code 1 highlights the main sections of the code that deal with the polling of the parallel port and the subsequent generation of external events from the parallel port to the model, only when there as a change in the state of the input register in the parallel port. This is done only when two conditions are met. The first one involves the non-existence of an event file, only when this field is left blank can the polling mechanism work. The second condition has been added to ECD++, and is the definition of another flag "-g" at runtime that instructs the simulator, through the function *isRealRun()* from the *loader()* class, that

the simulation being performed will use the memory space destined for the parallel port, which is usually protected by the kernel [59]. When executing the simulation with the "-g" flag, some portion of the code that request permission for the software to use restricted memory space is executed; once the request is granted it starts to execute an additional thread that will ultimately control the memory positions that change the assertion of bits in the parallel port.

```
MainSimulator &MainSimulator::loadExternalEvents( istream  &fileIn )
{
Root::Instance().initialize();

if (loader()->isRealRun()&&loader()->EmptyEventFile())
                            //real run and real events enabled
{
  try {
  tnow = Time::currentTime();
  if (tnow < Time::Zero) tnow = Time::Zero;
  deadline = Time::currentTime();
  deadline+=loader()->endeventTime();
  if (deadline < Time::Zero) deadline = Time::Zero +
      loader()->endeventTime();
  inportName = "in";  // the names of the ports are fixed
  outportName = "out";
  parvaluetemp =  pport.readfromParallel();
  if(parvaluetemp != parvalue){
  //values should be different to activate the event
     Port &port( Root::Instance().top().port(inportName) ) ;
     Port &outport(Root::Instance().top().port(outportName) ) ;
     convalue = parvaluetemp / 1.0;
     std::cout    << "Event occurs @: " << tnow.asString() << " "
                  << "Deadline @: " << deadline.asString() << " "
                  << "Value: " << convalue << "\n";
     Root::Instance().addExternalEvent( tnow, deadline,
                              port, outport, convalue ) ;
     parvalue = parvaluetemp;
  } else {
      //Do Nothing
      }
  } catch( InvalidPortRequest &e ) {
    e.addLocation( MEXCEPTION_LOCATION() ) ;
    throw e ;}
} else {... read external events from file...}
```

Program Code 1: Parallel Port read atomic DEVS implementation

The next Program Code 2 deals with the initialization and loading of the type of simulation, i.e. if the simulation is executed with the "-g" flag then the executable file will start a new thread and send the signals for the initialization of the motor, plus the loading of each external event as a single external event instead of a pool of events (if no event file is defined), this is done through a do...while structure.

```
MainSimulator &MainSimulator::run()
{
  if( !loader() )
  {
  MException e( "The MainSimulator loader not found!" ) ;
  e.addText( "The loader must be set before running the simulation.");
  MTHROW( e ) ;
  }
  if (loader()->isRealTimeRun()){//initialization of the motor
    pport.spinclockwise = false;
    pport.spincounterclockwise = false;
    pthread_create( &thread1, NULL, control_motor_, (void*) NULL);
    } //initialization ends
  loader()->loadData();
  DBG( "Loading Models...");
  loadModels( loader()->modelsStream(), loader()->printParserInfo() );

  Root::Instance().stopTime( loader()->stopTime() );
  startTime_m = elapsedTime();
 // run the following code at least once
do{
// at the end decide to contiune looping or not
  DBG("Loading ExternalEvents...");
  loadExternalEvents( loader()->eventsStream() );

  DBG("Running Root::Instance().simulate()...\n");
  DBG("startTime_m = " << startTime_m.asString());
  Root::Instance().simulate();
      if(loader()->isRealRun()&&loader()->EmptyEventFile())
            {if(Time::currentTime()>=loader()->stopTime()) break;}
} while (loader()->isRealRun()&&loader()->EmptyEventFile());
//real run and real events enabled
  loader()->writeResults();
  if (loader()->isRealRun())
{if(!pport.close()) std::cout << "\nCannot close LPT1 port!" <<
std::endl;}
  return *this;
}
```

Program Code 2: Main simulator code with real input capability. Added code in italics.

Whenever the simulation runs in *normal mode* (i.e. without the '-g' flag) the simulator loads all the external events from an external events (.ev) text file, and then the simulation is executed having a list of all the future external events in memory.

In the case where the inputs are changing in real time, it is not possible to anticipate future changes neither have a list of future timestamps that signal when the next event will take place. For this reason, whenever the simulation runs in real time, with real inputs, the simulator treats each external event as a single and unique external event, i.e. runs a complete simulation every time an external event is received. This whole approach takes considerable more time to execute than the normal execution, but considering the speed of the external events, this does not have major impact.

4.1. ECD++ with Hardware-In-the-Loop

The parallel port is tremendously slow for today's standards, and because there is a lot going on between readings, it is just not possible to have an accurate measure of the sampling frequency. One of the main reasons for this is that the code developed by the end user, the model developer, will run between readings. However, it is possible to measure the sampling frequency when there is no change in the input.

The sampling period measured on the platform was of 0.022 (s) which gives an approximate sampling frequency of 46 reads per second. As stated this number is only given as the empirical maximum frequency at which the simulator performs, any code developed by the user is executed between samples and will affect the sampling frequency parameter, making it much slower.

On the other hand, in the case of the 'writes' to the parallel port the contrary happens, because is very likely that the electro-mechanical interface, i.e. a motor, will be several times slower than the port frequency; therefore in some cases, there is the need to create delays between port-updates. For example, the original proof of concept of the output through the parallel port was to drive a small 2-coil stepper motor; the type of motor commonly found in toys, CD-ROMs and hard disk drives, for spinning this type of motor a defined sequence of switches (bits) need to be turn-on and off and a delay needs to be introduced between changes to accommodate the system's speed to the motor. In the final implementation the time between updates to the motor is done in the user software or it can be done in the model, because the motor used is a brushless DC motor that only needs one switch (bit) for each direction.

4.2. Motor Driver

The main difference of using a stepper motor is that the speed, spin direction and position of the rotor can be controlled with trains of pulses that can be easily generated by a computer, while a continuous brushless DC motor requires polarity inversion to switch the direction of spin (which is done via hardware), and to position the rotor accurately requires slightly more elaborated electronics. Despite this consideration, a single general driver can be built for both types of motors. For an initial test setup, in a four-wire stepper motor the coils can be connected in such a way that every time that they are energized with a predefined set of binary numbers the rotor spins ¼ of a turn, therefore by keeping track of any number in the set is possible to know the position of the rotor.

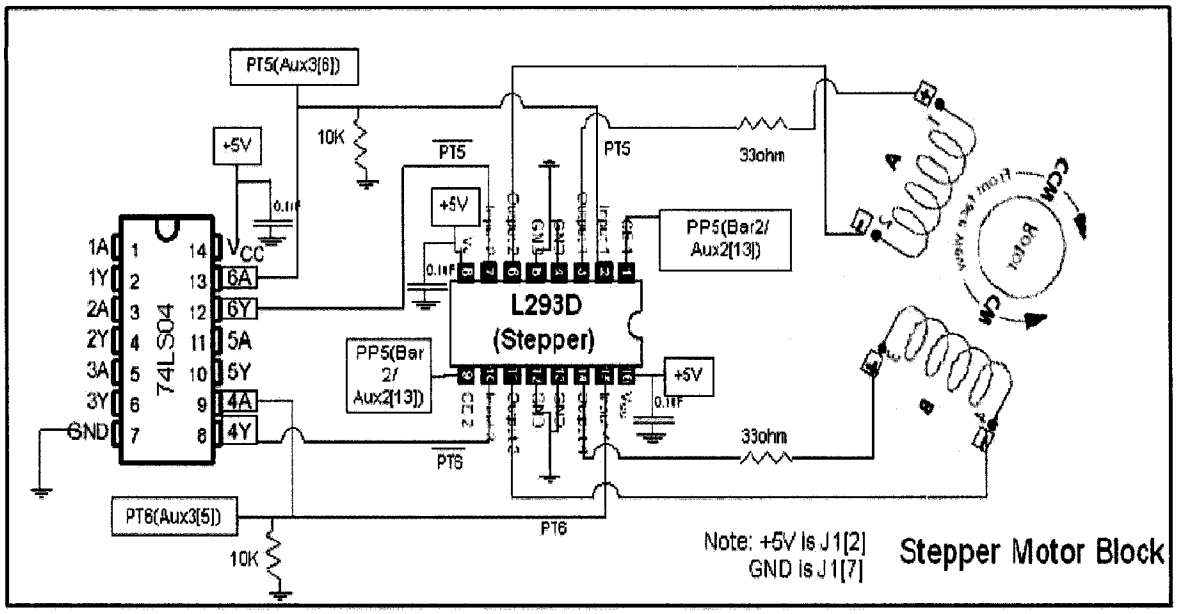

Figure 15: Stepper motor test circuit layout [60]

According to the schematic shown in the Figure 15, three input-bits are required for the motor to spin: two of them are connected to the motor through a quad push-pull driver and the other is connected to the enable pin of the same driver to enable the outputs of it. The logic inverter is only used to minimize the wires coming from the computer to two. All the capacitors are in place to limit the ac-ripple on the dc power source. The low value resistors limit the current that is fed to the motor and the high value resistors are set in a pull-down configuration. Figure 16 shows the circuit mounted on a breadboard.

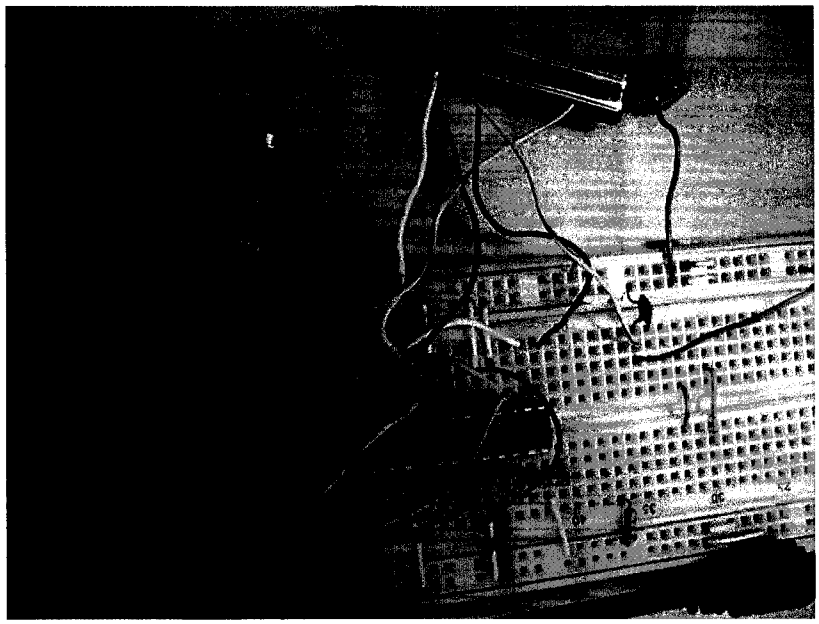

Figure 16: Stepper Motor test circuit

Since the simulator is performing multiple different tasks and because the actual drive of the motor implies outputting a value, wait for a considerable time and output a new value according to a predetermined sequence. An easy implementation of these control system would assume that a specific processor is dedicated only to generate the write-and-wait sequences, while it communicates with a different control processor that sends enable and control signals. A good abstraction of this concept makes use of a new thread that only runs the motor and leaves the main simulator thread ample time and flexibility to do any kind of control it needs to do without having to make major changes to the architecture of the system.

4.3. Implementation

During the development of this test case, the new IDE environment was used and proved useful in the debugging and optimization of the project. Additionally, the information

presented on the IDE made the design process easier and much quicker than a purely text based environment.

The control of the sequence of steps for the motor can be done through a new thread that behaves as a completely isolated processor, which its only task is to generate the steps and delays required for the motor to spin. This is depicted in Figure 17.

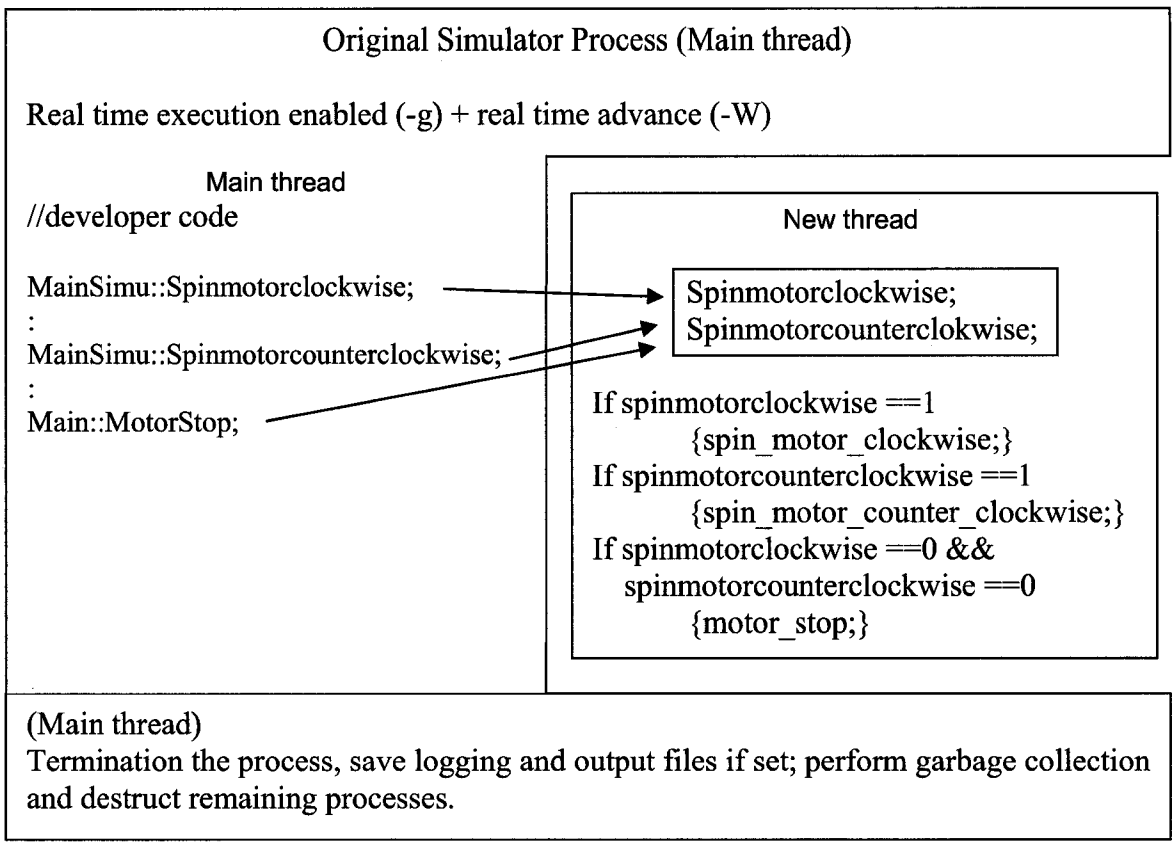

Figure 17: Spin motor thread implementation – pseudo code

Figure 17 shows that when 'real simulation' (when the "-g" flag) is selected, the simulator creates a new thread and defines shared variables in it; it is through these variables that the main thread is capable to control the execution of the code required for the motor to spin in either direction. The slave thread reads the values of these variables in an infinite loop, and

then uses them as flags to execute defined sequences of writings to the Parallel Port output register. For a stepper motor four updates are required, and occur after a short delay and then are repeated indefinitely until there is a change in one of the control variables. For a DC motor the updates are written in every execution of the thread, this is obviously not necessary since it is possible to set the register just once to keep the motor running, but then the flexibility of having a second thread is lost. Additional care has been taken to avoid unknown states, i.e. both control bits asserted, hence, the implemented condition for the motor to spin is to have only one of the Boolean variables asserted (true) while the other is unasserted (false).

The implementation of writing to the Parallel port and generating the sequence that moves the motor can also be cast as a DEVS atomic model:

$$\text{Spin Motor} = \langle X, Y, S, t_a, \delta_{int}, \delta_{ext}, \lambda \rangle$$

***X*:** Parallel port external events: these are the changes in the state of four inputs:

- Spin_Motor_Clockwise, Spin_Motor_CounterClockwise, Turn_Left, Turn_Right

***Y*:** output port: is the Parallel port.

***S*:** system states:
- Spin Motor Clockwise,
- Spin Motor Counter Clockwise.
- Stop motor.
- Turn Left.
- Turn Right.

t_a: time advance function: handled externally from the simulator.

δ_{int}: internal transition function: not required for this implementation.

δ_{ext}: external transition function: checks for changes on either one of the control bits.

λ: output function: Writes a predefined 3-bit data from a pool of values to the Parallel port depending on the state of the control variables.

The implementation of the initialization and closing of the parallel port can be easily taken from the explanation from above by first requesting permission to the system to access the memory space destined to the parallel port. The Program Code 3 shows how such scheme is implemented.

```
bool parallelPort::setup(void){
if(ioperm(DATA,3,1)) return (0);
// if access granted initialize DATA to 0x00
outb(0x00,DATA);
//initialize all control pins to low (c0, c2 and c3 are inverted)
// c0 being the LSB
outb(0x0B, CONTROL);
// return 1(true) is successful
return (1);
}

bool parallelPort::close(void){
//set pins to 0 and set control pins to low
outb(0x00,DATA);
outb(0x0B,CONTROL);
//Remove access permision to 3 I/O addresses (DATA, STATUS, CONTROL)
//starting from the DATA address
if(ioperm(DATA,3,0)) return(0);
//if successful return true = 1
return (1);
}
```

Program Code 3: Parallel port setup and termination

Once the proof of concept test was successfully finished, a more complete test platform was required to test different scenarios. Small carts are widely used in the manufacturing plants, warehouses and almost any industry in general, as transport vehicles for the relocation of goods in short distances. In a very simplified automation scheme, we would be interested in making this carts change direction when they sense some obstacle in their way. Based on this concept, and to provide a complete implementation of the RoboCart system, a LEGO

NXT Robotic Kit was acquired, and a small cart was built. The advantage of this *prototyping* tool is that it provides all the electro-mechanical support required by small to medium proof-of-concept projects and small prototypes. The standard NXT kit comes with three DC motors and sound, touch, infrared and temperature sensors plus a special 'brick' that contains a microcontroller and electronics capable of receive information from the sensors and drive the motors. For the test case, the basic robotic cart was assembled without the controller brick, and the motors connected to the parallel port through a slightly modified version of the circuit used to drive the stepper motor.

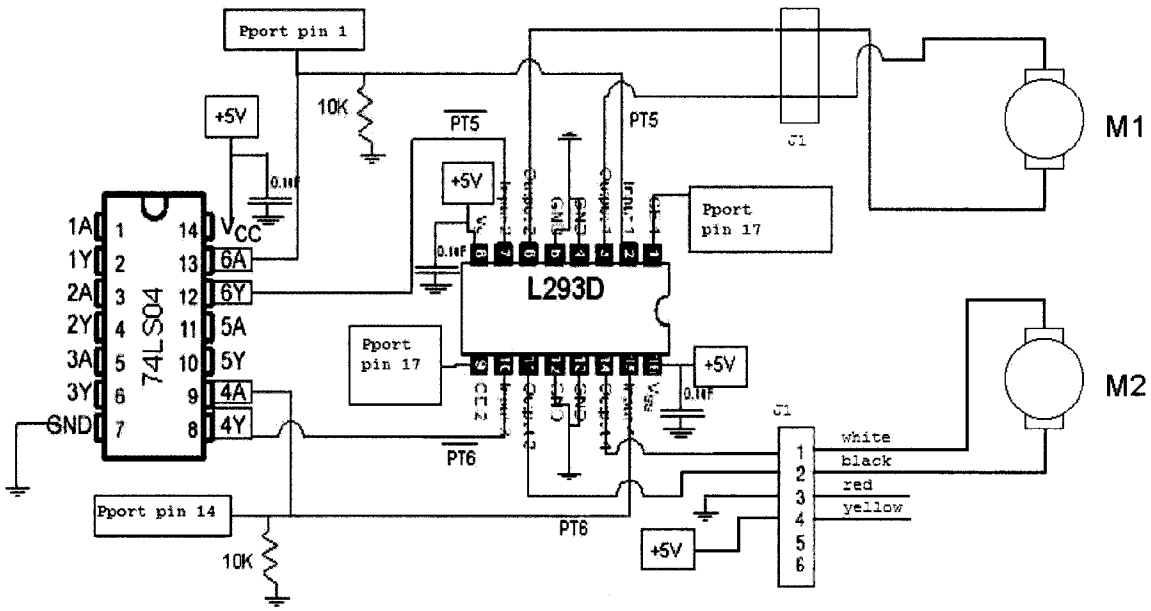

Figure 18: Modified DC Motor Driver Circuit

From the modified version of the driver circuit, in Figure 18, the same motor driver that controls one stepper motor is used to control two DC motors, the synchronization is done via software. The spin direction of each motor is controlled by one bit, C0 controls the left

motor and C1 the right motor, and the whole system is turned on or off by asserting a third *enable* bit (C3), which can also be used to brake.

In the software side of the implementation, the Parallel port atomic block acts as a driver providing the required code to initialize the Parallel port for subsequent use, and it closes the port when the program finishes by calling a `close()` function that restores all port outputs to zero. The Parallel port block was created as a separate class and is called from the new thread, this way it is easier to make modifications, i.e. change stepper or dc motors, or upgrade the control algorithm in future developments; i.e. if there is the need to change the controlling method to some other control algorithm, then the class file is the only one that needs to be changed.

The excerpted code in 4, shown above demonstrates the use of the methods used to spin the motor so that the cart moves forward and backward. The methods to turn left and right are set in a way that minimum resolution for turning is 90 degrees, this because the constructed prototype is only capable of sensing obstacles with the only one push-sensor available in the kit, located at the front of it. Therefore, to avoid completely crashing into an obstacle constantly, the cart rotates 90 degrees to position itself parallel to the obstacle and continues rolling forward.

```
void parallelPort::output2P_CONTROL(int cValue){
      int cV = cValue;
     if(spinclockwise&&!spincounterclockwise){
       outb(0x03,CONTROL); // for DC motors this is enough
      //if it's a stepper motor uncomment these lines
/*    delay(cV);
      outb(0x02,CONTROL);
      delay(cV);
      outb(0x00,CONTROL);
      delay(cV);
      outb(0x01,CONTROL);
      delay(cV); */      }

     if(!spinclockwise&&spincounterclockwise){
       outb(0x00,CONTROL); // for DC motors this is enough
      //if it's a stepper motor uncomment these lines
/*    delay(cV);
      outb(0x02,CONTROL);
      delay(cV);
      outb(0x03,CONTROL);
      delay(cV);
      outb(0x01,CONTROL);
      delay(cV);*/
      }

     if(!spinclockwise&&!spincounterclockwise)
       outb(0x08,CONTROL);

      if(turn_left&&!turn_right)
       {
       outb(0x02,CONTROL);
       delay(150000000);
       turn_left = false;
       }

     if(turn_right&&!turn_left)
       {
       outb(0x01,CONTROL);
       delay(150000000);
       turn_right = false;
       }
      //end of parallelPort::output2P_DATA
```

Program Code 4: Motor spin driver - parallel port

The turn-left and turn-right methods are implemented in such a way that whenever they are summoned they restore the control bit automatically after enough time to complete a 90-degree turn to either side.

The model of the robotic cart is rather simple and can be better explained by the use of a diagram like the one in Figure 19. When the controller receives an input form the touch sensor, meaning that the cart is facing an obstacle; the control code moves the RoboCart backwards to have more space for taking the turn. Due to the availability of a single sensor the turning is done alternating the direction of the turn (i.e. the RoboCart turns either side twice to the left, but once to the left and the next one to the right); by increasing the number of sensors or, even better, having the provided ultrasonic sensor would make the direction decision more accurate, but the system's software driver would have to increase in complexity, because this sensor uses the Inter-Integrated Circuit (I^2C) communication protocol. Therefore, the minimum turn that the RoboCart can take to be completely sure that it is perpendicular to the obstacle is 90 degrees. The turning is done by spinning the wheel of the turning side clockwise while the other wheel is rotating counter clockwise.

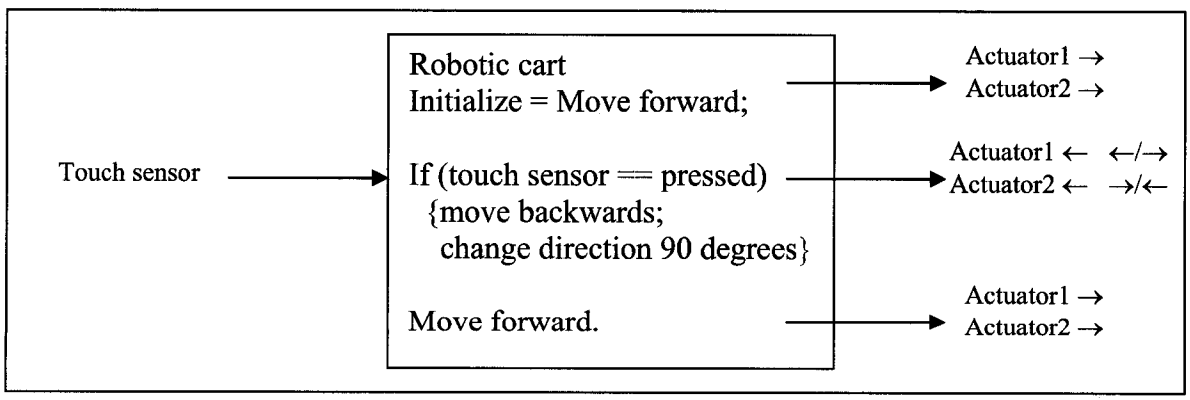

Figure 19: Robotic Cart – pseudo code model

This behaviour can be easily represented by a CD++ model. The C++ code for the RoboCart is shown in the Program Code 5.

```
/*******************************************************************
* CLASS RTPport
*******************************************************************/

/*******************************************************************
* Function Name: RTPport::RTPport()
* Description: Constructor
*******************************************************************/
RTPport::RTPport( const std::string &name ) : Atomic( name )
, in( addInputPort( "in" ) )
, out( addOutputPort( "out" ) )
, preparationTime( 0, 0, 0, 1 )

{
std::string     time(     MainSimulator::Instance().getParameter(     description(),
"preparation" ) ) ;
//we can get some parameters that we might need form the model file
       if( time != "" )
       preparationTime = time ;
       MainSimulator::Instance().Spin_Motor_Clockwise();
}

/*******************************************************************
* Function Name: RTPport::initFunction()
* Description: Initialization Function
*******************************************************************/
Model &RTPport::initFunction()
{
       ackNum = 0; // to recover the input value from the external event
       return *this ;
}

/*******************************************************************
* Function Name: RTPport::externalFunction()
* Description: External Function handler
*******************************************************************/
Model &RTPport::externalFunction( const ExternalMessage &msg )
{
       ackNum = static_cast < int > (msg.value());
       if (msg.value()==216) { //checks if the external event comes from the touch
sensor
       MainSimulator::Instance().Spin_Motor_CounterClockwise(); //move back for
           holdIn( Atomic::active, preparationTime ); //the preparationTime from
the model file
       }
       else passivate(); //if not go to sleep
       return *this;
}
```

Program Code 5: Model file.

In the above Program Code, the initialization function sets the motors to start spinning forward, and initializes some intermediate variables. If an external transition function is triggered by the touch sensor (an external event is generated), the RoboCart model moves backwards for a certain time-period given by the time advance function entry in the model file (.ma). This also triggers the internal transition function (ITF), which is activated after the time advance function has elapsed, the code inside the ITF turns the RoboCart to a different side based on the last direction of the turn. Finally the model continues moving forward (the component goes to rest) and waits for an external event that indicates that a new obstacle has been found and an evasive action is required to overcome such obstacle. This is presented in the Program Code 6.

```
/*******************************************************************
* Function name: RTPport::internalFunction()
* Description: Internal Function handler
*******************************************************************/
Model &RTPport::internalFunction( const InternalMessage & )
{
test = 1^test; //ex-or toggles the bit, thus 'remembering' the last turn
// and turning in the opposite direction.
if (!test) MainSimulator::Instance().Turn_V_Left(); //turn left or right
if (test) MainSimulator::Instance().Turn_V_Right();
MainSimulator::Instance().Spin_Motor_Clockwise(); // move forward again
passivate(); //go to sleep
return *this ;
}

/*******************************************************************
* Function Name: RTPport::outputFunction()
* Description: Output function handler - writes info about time and events
*******************************************************************/
Model &RTPport::outputFunction( const InternalMessage &msg )
{
      sendOutput( msg.time(), out, ackNum) ;
      return *this ;
}

RTPport::~RTPport()
{
      // N/A
}
```

Program Code 6: Model file (cont)

Also in the Program Code 6 the output function records or displays, depending on the selection mode of the simulation, the time at which the external event was received and the value of the external even, this is last value is very useful because, if multiple sensors are used then we can tell the controller the direction to which it should go. If a more advanced sensor is used (i.e. ultrasonic sensor) then many measures can be taken from many directions to find which way presents the least obstacles. The cart built for the project and the driver circuit can be seen in Figures 20, 21 and 22.

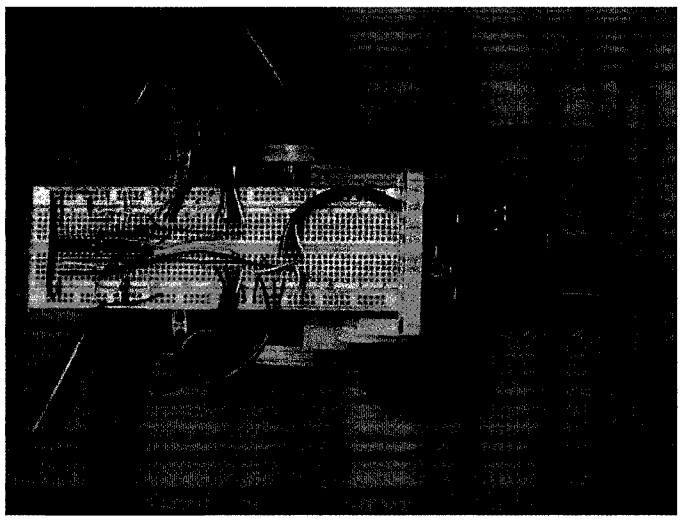

Figure 20: RoboCart top view

Figure 21: RoboCart – side view

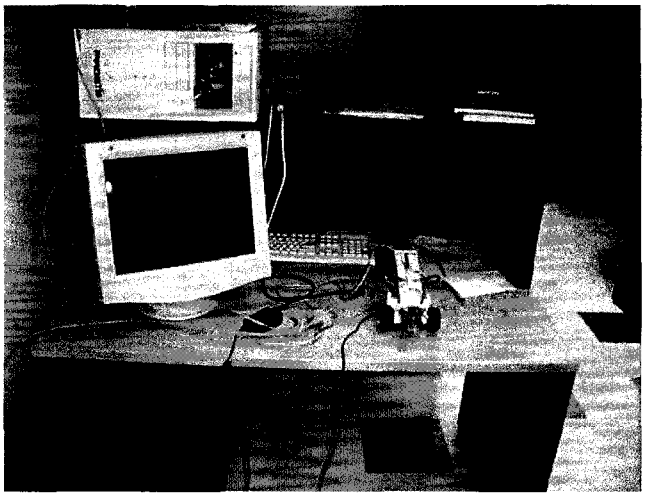

Figure 22: RoboCart – System view (AMPRO board inside CPU case, Monitor not used)

Initially the RoboCart was tested with the external events coming from an event file with random times and event values, some of them spread in time and some others were closely spaced in time. The events in the file that were close together created confusing behaviour in the RoboCart emulation, mostly because the event generation was too fast for the dynamics of the motors of the RoboCart and the time limits of ECD++ could not be met, sometimes this led to a random behaviour and the parallel port remains enabled even after the termination of the program.

For sufficiently spaced external events the results are the expected ones, when an external events occurs the RoboCart moves backwards according to the time defined in the model (measured in seconds) to allow sufficient space to turn, then the internal transition is fired and the RoboCart turns right or left based on the last turn, finally it displays or prints the event information to the screen or file and moves forward. This process is repeated for all external events coming from the event file.

When running the simulation in *real mode*, i.e. receiving input form the parallel port and generating an external event as soon as a change in the input register of the parallel port is

detected, the RoboCart behaves as expected, moving back and then turning to either side depending on the last turn. However, the touch sensor is simply a mechanical switch that switches between high impedance and ground, to keep the circuit simple no provisions were made for the *bouncing* effect typical of this switches. Whenever the repetition of changes of the input register is too fast (i.e. artificially pressing and releasing the sensor very fast), the system can keep up with this repetition and the same erroneous behaviour is present, that is to say that the port stays enabled even after the termination of the program.

For a normal execution of the *simulation*, a small maze-like path was constructed, the RoboCart was placed at the beginning and the program executed. The RoboCart went through the maze and every time it hit a wall the control algorithm developed in CD++ took control of the situation; forcing the RoboCart to move backwards for the time given in the model, then turn to a wall free side based on the last turn and move forward again until a new obstacle pushes the touch sensor, when the process is repeated.

To develop the Development Environment tool further, it is possible to create a performance measure that can be used to compare the speed and accuracy of different versions of CD++ against other DEVS-based simulators. Such comparison can be based on the *benchmarking* of the performance of the simulators when running similar simulations of identical models. With the information provided by the benchmark and the use of common debugging tools it is, also possible to find the cause of performance problems of one simulator versus another. Once the cause of any problem is found, multiple solution strategies can be analyzed, and the simulator under test can be improved. The creation of the benchmark and the strategy of analysis are discussed further in the next Chapters.

5. DEVStone

DEVStone uses the well-known synthetic Dhrystone Benchmark [61]. This benchmark measures the time that takes for a series of basic integer operations to be executed; given the discrete nature of the models to be simulated with CD++ and integer benchmarking technique was obvious.

The synthetic benchmark developed for Parallel CD++, named DEVStone, is explained in [16]:

> "*... we created a synthetic model generator (which we called DEVStone), which produces a variety of models with diverse structure and performing a mix of common operations. We focus in the aspects of the models that have impact on performance, namely size of the model and the workload carried out in the transition functions.*"

Such aspects are artificially generated by a synthetic model generator (written in PERL) that lets the developer chose the depth of a set of coupled models and the width of *Real-Time Atomic* models within each coupled model, the interconnections among each model are given by four fixed schemes, 3 original and the new highly coupled model:

- **LI** models, with a low level interconnections for each coupled model,
- **HI** models with a high level of input couplings, and
- **HO** models with high level of couplings.
- **HOmod** models, which are the new complex models with very high level of couplings.

The new model was added because the quality of the results obtained in the original DEVStone was non-indicative of the expected performance measures of the benchmark. Therefore, we decided to improve this benchmarking tool and use it to compare the performance of ADEVS and CD++ through the simulation of various types of models.

DEVStone models use two key parameters: *d* the depth and *w* the width, with which a DEVStone model of any given size can be implemented, where each depth level, except the last, will have *w-1* atomic models, and each atomic model provides customizable Dhrystone running time. The inner model of such scheme is comprised of a 'coupled *atomic* model' that embeds a single 'atomic model'; hence, the total number of atomic models that the model will have is given by:

$$\#RTAtomic \text{ models} = n = (d-1)*(w-1)+1 \qquad (1)$$

Theoretically, by knowing the connection and interconnection patterns among the modules it is possible to compute the total simulation time of a given model assuming that the passing of messages is practically instantaneous when an external event is triggered, this in turn triggers the external transition function of each atomic model and consequently an internal transition function is also scheduled [62]. By saying this, we can be sure that the total number of transition functions is twice the number of atomic components in the model, in other words, every atomic model has an internal and an external transition function and each transition function represents one atomic model.

$$\#RTAtomic \text{models} = n_i = \#\text{internal} \quad \text{transitions} \qquad (2)$$
$$\#RTAtomic \text{models} = n_e = \#\text{external} \quad \text{transitions} \qquad (3)$$

The model can be conceived as a coupled component that wraps *w* atomic components and another coupled component, which in turn has a similar structure. The connection with the exterior is done by one input and one output links. The input feeds the first coupled

component; the coupled component then builds links from the single input each of its subcomponents. A graphical description of the blocks that build a LI model generated by DEVStone is given in Figure 23.

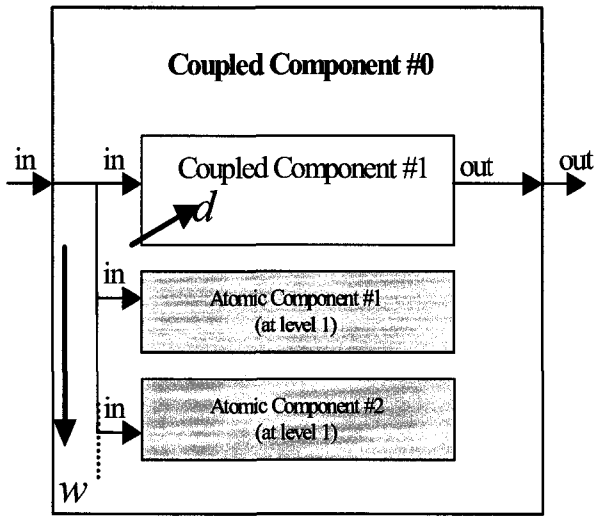

Figure 23: DEVStone LI model

The inner component of the model will just have a coupled model that will function as a wrapper for a single atomic model; this is shown in Figure 24.

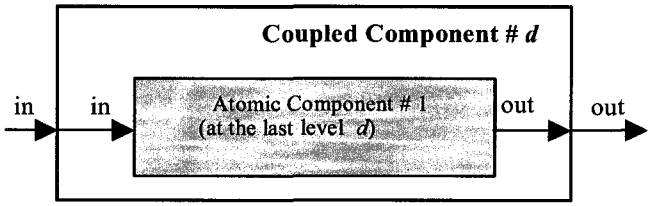

Figure 24: DEVStone inner 'coupled atomic model'

The difference among the three predetermined interconnections types, LI, HI, HO resides on the quantity of internal and external couplings for each level. LI types are as the models presented in the graphics with just one external input and one external output coupling. HI and HO types have more interconnections among the components and the outputs respectively. HI type of model connects the output port of an atomic block *i* to the input port of the next atomic block *i+1*, this is depicted in Figure 25.

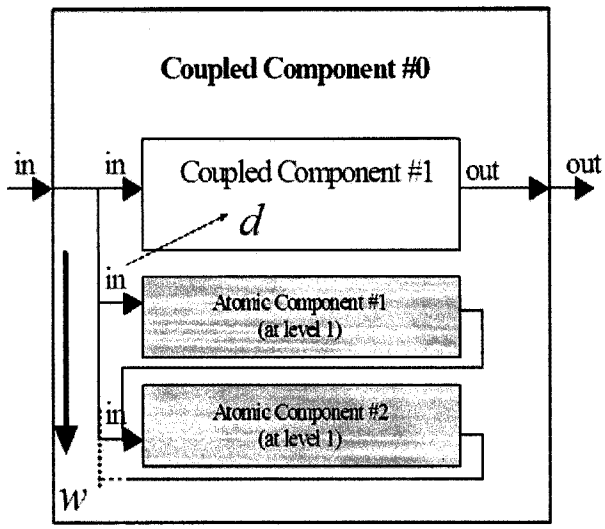

Figure 25: DEVStone HI model

This in turn generates an increase of external events going to every atomic model given by:

$$\#\text{InternalTransitions} = n_{it} = \big((w-1)+(w-2)\big)*(d-1)+1 \quad (4)$$

$$\#\text{ExternalTransitions} = n_{et} = \big((w-1)+(w-2)\big)*(d-1)+1 \quad (5)$$

The HO type has two input ports and two output ports in each coupled model. The second input port in the coupled component is connected to its first atomic component. This atomic model connects its output to the second output of its parent, thus, the resulting number of interconnections is similar to the HI type of model –equations (4) and (5) are identical for this type– with the difference that there are more inter-messages between levels in HO types than in the HI type of modes, this can be seen in Figure 26. These equations does not take into account the time that is required in reality for inter-message passing among atomic models, coupled models, the simulator through the coupled coordinator and the root coordinator.

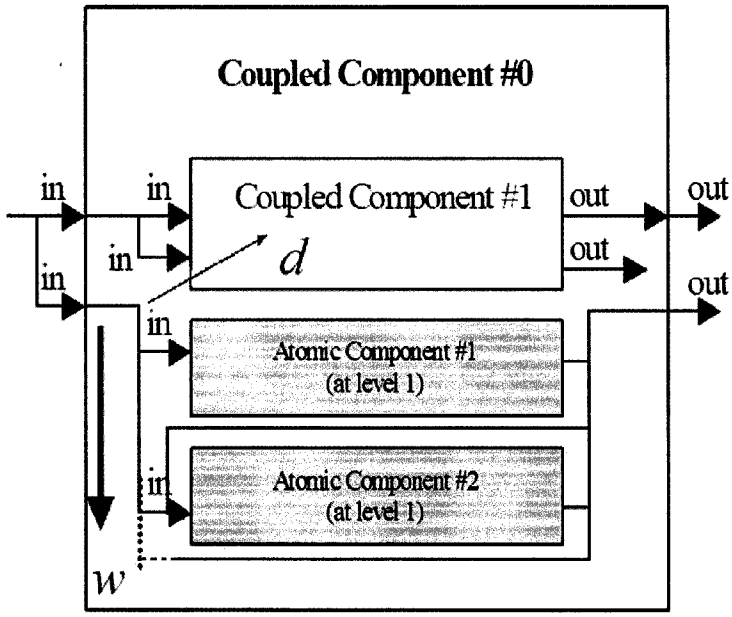

Figure 26: DEVStone HO model

Finally, a new model was developed that increments the message traffic and exponentially explodes the interchange of messages among coupled models. **HOmod** models have a second set of *(w-1)* models where each one of the atomic components triggers the entire first set of *(w-1)* atomic models. These in turn have their outputs connected to the second input of the coupled model within the level. With such interconnections, the inner model receives an amount of events that has an exponential relationship, given by equations (6) and (7), between the width and the depth at each level. The **HOmod** model is shown in Figure 27.

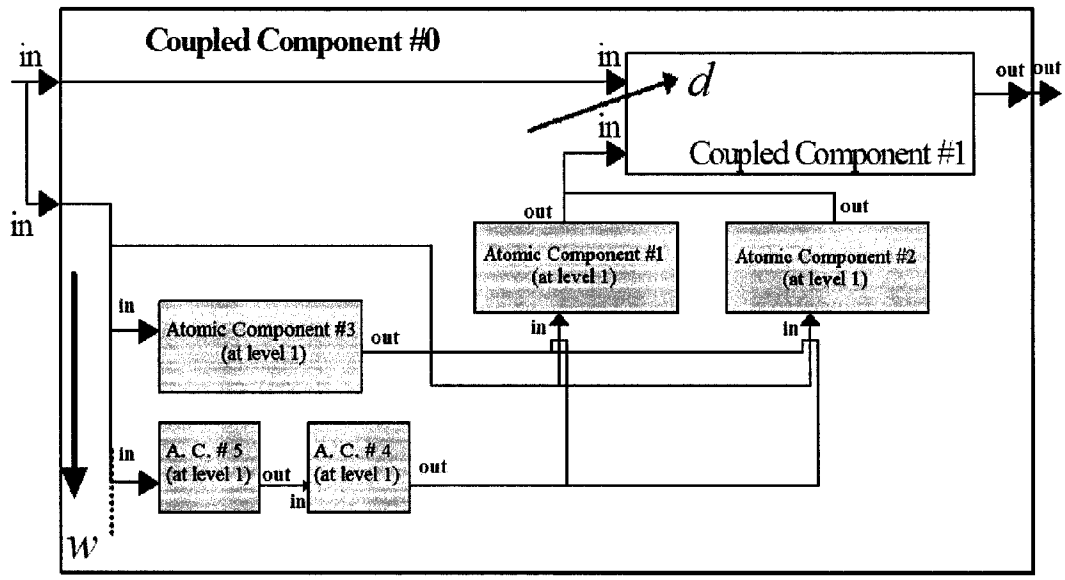

Figure 27: DEVStone **HOmod** model (shown explicitly for $w = 3$)

In addition, the equation that rules the behaviour of the **HOmod** Model is given by:

$$\text{\# Internal Transitions} = n_{it} = (w-1)^{2*(d-1)} + \left(2*(w-1) + \sum_{1}^{w-2} i\right)*(d-1)+1 \quad (6)$$

$$\text{\# External Transitions} = n_{et} = (w-1)^{2*(d-1)} + \left(2*(w-1) + \sum_{1}^{w-2} i\right)*(d-1)+1 \quad (7)$$

5.1. *DEVStone Implementation*

Based on the DEVStone benchmark tool (originally developed to measure the performance of Parallel CD++), the extended DEVStone functionality was implemented and, a modified version of the original was made available for the standalone version of CD++, which runs in a command-line environment in Linux; as well as a version for ADEVS, which also runs in a command-line environment in Linux. The implementation paid particular attention to maintain the philosophy of the original benchmark tool separating as much as possible the

model from the simulator, giving the user/programmer flexibility in the use of the new benchmarking functions.

For CD++, DEVStone was implemented as an automatic model generation tool based in a common template for each model. The coding was done in PERL, and it consists of a file per each model. Based on a common template for every coupled component per model, each PERL script uses multiple loops to form the final model using the desired depth and width, DEVStone atomic components are embedded in CD++.

The coupled components of any DEVStone model are defined in the model, and generated following the procedure described above. However, the CD++ simulator contains a line where all the components of a level (the *width* of the level) are declared. For widths of more than 1839 DEVStone atomic components inside a level, the simulator generates an error stating that the line where the components are declared is too big. To overcome this problem, and for widths larger than 1500, DEVStone splits the width-components in multiple lines of 1500 atomic blocks. With this patch, the limits of width for the models are given solely by the availability of computing resources.

The atomic component of DEVStone needs to be implemented as part of a registered atomic component inside the CD++ simulator. The implemented DEVStone atomic model can be expressed as a simplified DEVS model:

$$DEVStone\ Atomic\ component = \langle X, Y, S, t_a, \delta_{int}, \delta_{ext}, \lambda \rangle$$

- ***X***: Input port: in.
- ***Y***: output port: out.
- ***S***: the Atomic component has no internal states.

t_a: the time advance count is provided internally.

δ_{int}: internal transition function: Number of Dhrystones to execute, this data comes form the model at runtime, once executed the model goes to passive mode. It is always activated after an external transition.

δ_{ext}: external transition function: Number of Dhrystones to execute, this data comes form the model at runtime. It is activated by an external event.

λ: output function: print relevant information for debugging: time, value and port.

On the other hand, the DEVStone benchmark model developed for ADEVS has a similar Atomic component structure as the one developed for CD++. However, due to the rather tight integration between the modeling environment (C++) and the simulator (C++ libraries), some challenges arise at the time of the implementation. One of them relates to the generation of the coupled and atomic components, since each one comprises a unique C++ file, meaning that for any model to be created in the ADEVS version of DEVStone new code needs to be created in C++. Another important problem that is generated from the first problem deals with multiple C++ and libraries that are created automatically and the way they need to be compiled and linked.

Since there is, no more separation of atomic and coupled components both of them need to be created at the same time as individual C++ files. To overcome this problem DEVStone for ADEVS was coded as PERL scripts for each model type as well as in the CD++ version. The scripts contain loops that based on a template per each model type and the desired width and depth generate a *component* file and a *component* library file, for each atomic and coupled component. The linking among components is done in the coupled component files and the atomic component model and library files are mainly used as libraries (only the names of variables and component labels are changed). Still, ADEVS does not provide exactly the same functionality of CD++, it can read data at runtime but it

cannot use this data as an internal parameter, this is problematic with the input of the number of Dhrystones in the internal and external transition functions. As a result, the number of Dhrystones is entered in the creation of the simulator as a library file by the PERL script. The final external linking of the model and the creation of the simulator are done in an additional file that is also created by the generator by keeping track of the outer coupled and atomic components of the model. A *makefile* script is also generated by the PERL scripts to automate the compiling and linking of all the C++ files.

6. DEVStone Results

The main objective behind the implementation of DEVStone is to provide a tool that can be run in all the hardware platforms where our CD++ simulator can run. The benchmark purpose is to be a self-contained tool to measure the performance of different implementations of DEVS simulators for different hardware or the performance of different versions of DEVS simulators that might integrate new functionalities but that would not be necessarily 'better suited' for certain purposes.

In any case a parameter of comparison needs to be defined to compare future performance metrics with an *original test environment;* with this in mind and considering that future developments of CD++ will involve the use of a general purpose Personal Computer with whatever configuration is standard for the time, we chose to setup the *original test environment* with the following commercial grade characteristics:

CPU: Pentium 4 Dual Core @ 3.2GHz (800MHz FSB, HT, 1MB L2)
Chipset: Intel 950G
RAM: 2 GB (4 x 512MB DDR2-533)
Hard Drive: 80 GB physical - 35 GB ex3 Linux partition (7,200 rpm, SATA)
OS: Fedora Core 6

Both simulators were compiled in a second 'developer' system using the Eclipse-based CD++ Builder for CD++ -compiled with GCC 2.95.3- and the GNU Compiler version 3.4.2, without debugging information and standard (default) compile-and-link options for ADEVS.

Following the setup set by the original DEVStone the simulations respond to 10 predefined constantly spaced external events, therefore the theoretical time calculated for each simulation can be found by using equations (1), (2) and (3) for Li models and (4), (5) for HI or HO models, and is given by:

$$\text{Theoretical Time} = \begin{cases} (n_e * \delta_{\text{int}} * ev) + (n_i * \delta_{ext} * ev) & (8) \\ (n_{et} * \delta_{\text{int}} * ev) + (n_{it} * \delta_{ext} * ev) & (9) \end{cases}$$

Where:

n_e = Number of external transitions

n_i = Number of internal transitions

δ_{int} = Time spent in an Internal Transition

δ_{ext} = Time spent in an External Transition

n_{et} = Number of external transitions for HI, HO models

n_{it} = Number of internal transitions for HI, HO models

ev = Number of external events = 10

Since the measuring mechanism implemented in CD++ and ADEVS has a minimum resolution of ±1 (s), or a span of 2 seconds, for comparison purposes, only points that carry enough information were considered as valid performance points. In other words, the minimum measured time to be considered valid in the experiment is at least 3 ± 1 (s) because it carries at least 50% of valid information: a region of 2(s) valid time and a region of 2(s) of uncertain information. In some cases where the measured values are below this mark but the results are deemed important, the values are considered in the research but future researchers should considered that the uncertainty of these values is extremely large.

Running the simulations, we found some general limitations to the experimental setup:

- The minimum depth and minimum width for any model generated by DEVStone for CD++ or ADEVS is 2.

- The depth cannot be greater than 195 levels, this due to the GCC compiler, which finds too many nested loops inside the executable of the ADEVS simulator. CD++ doesn't seem to have a limit on the depth, 4000 levels by 3 components per level were read by the CD++ simulator without a problem, however there is a caveat: the test that we ran only assures that the initial setup is performed, in other words that the simulator reads the model without reporting any errors, before running any simulation.

- In most extreme cases, it is possible to initialize the simulation, i.e. 195 x 1839 and run it to 00:00:00:00 or 0.0 for ADEVS, but it is not possible to run the simulation to any time longer than 0. In CD++, an 'unexpected error' message is displayed or the process is killed by the Out-Of-Memory (OOM) kernel service; and in ADEVS, the simulation is killed by the OOM kernel service as well. It seems that whenever the simulator requests massive amounts of memory to the operating system, beyond the available physical memory and some of the virtual memory, Linux decides to terminate a potentially harmful process, although no further investigation on this subject was performed.

We started tackling the benchmark measuring the initialization time for CD++ and ADEVS. We selected a nominal value to start measure the initialization time: since the depth limit is set by ADEVS with 195 levels of depth it is just natural to select this value; but the width does not seem to have an upper limitation, therefore for the initialization setup we decided to select the width value as the width value that could fit a line in CD++, nominally we set the width to 1839 components.

For the LI model the outcome of the initialization test was:

depth =	195
width =	1839
δint =	1.0 (ms)
δext =	0.1 (ms)
T_{sim} =	0 (s)

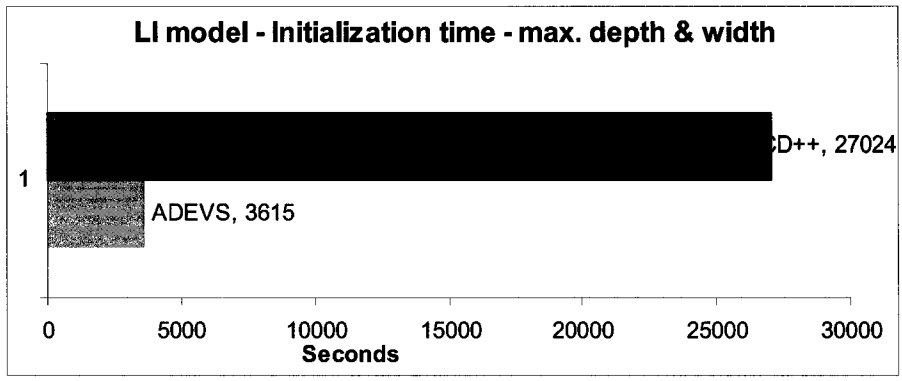

Figure 28: DEVStone Initialization time – LI model

We consider the initialization time for this purpose as the required time for the compilation of the source code and the initial execution of the executable to setup the model before simulating it, i.e. the simulation time to $T_{sim} = 0$. The compilation time of CD++ for big models is negligible; because the simulator and the model are two completely separated entities the compilation of the simulator takes only a couple of minutes and can be used for any model, in our case the simulator was compiled only once and reused for all the simulations in this report. On the other hand the compilation time is taken into account, only for initialization purposes, for ADEVS mainly because any change in the model involves a new compilation of the source code. Therefore, for small to medium sized models where a minor correction of the model was required the compilation time surpassed the simulation time. This clearly affects the performance of the simulator whenever a slight change needs to be made in the simulator; as any change involves the compilation of the modified source code before execution.

From the graph, it is obvious that ADEVS outperforms CD++ in terms of initialization speed, even when the compilation time is considered for the former. To see what is happening with CD++, a second run was called for, but this time we used the profiling

option during compilation and linking; unfortunately this is an 'invasive' method and makes the execution of the simulation much slower with a total simulation time of 82897.48 (s).

Table 1: . Profiler output. Flat profile: for the simulation of LI models

% time	cumulative seconds	self Seconds	Calls	self Ks/call	total Ks/call	Name
29.09	4633.2	4633.2	1427848	0	0	ProcessorAdmin::processor (basic_string<...> const &)
17.91	7486.26	2853.06	450903011	0	0	basic_string<...>::compare (basic_string<...> const &, unsigned int, unsigned int) Const
15.86	10012.31	2526.05	162000611	0	0	basic_string<...>::rep(void)const
9.59	11539.19	1526.88	3307266338	0	0	basic_string<...>::length(void) const
7.91	12798.64	1259.45	989592590	0	0	basic_string<...>::data(void) const
7.57	14004.23	1205.59				String_char_traits<char>::compare (char const *, char const *, unsigned int)
6.85	15095.7	1091.47	287281638	0	0	Processor::description(void) const
5.05	15899.58	803.88	1065955666	0	0	basic_string<...>::Rep::data(void)

According to the initial result of the profile most of the initialization time, 20.09 % of it, is spent somewhere inside the processor block. Other important source of delay during initialization seems to be the search and compare of symbols done by C++ libraries, which are performed while loading the model.

A more thorough analysis is possible, with the help of the GNU Profiler that comes standard with the GCC compiler, which in turn comes with any major distribution of Linux. This analysis provides a *Call Graph* that indicates functions inside the program and the *relative* time that the computer takes running those functions. The caveat here is that the percentage of time spent in each function is not an *absolute* time, i.e. the sum of the percentages will not yield 100%. This time is relative to the total running time but also to the time spent inside the 'parent function' [63].An edited list is provided in the appendices and stripped down parts of it are provided next.

Table 2: Call Graph of the CD++ Simulator

Granularity: each sample hit covers 4 byte(s) for 0.00% of 82897.48 seconds

index	% time	Self (s)	Children (s)	Times called	Function name
		0.00	81691.17	1/1	Main [1]
[2]	98.5	0.00	81691.17	1	MainSimulator::run(void) [2]
		0.00	79354.02	1/1	MainSimulator::loadModels(istream &, bool) [3]

		...	...	...	...
[4]	85.4	2853.06	67920.51	450903011	Basic_string<...>::compare(basic_string<...> const &, unsigned int, unsigned int) const [4]
		1496.24	50534.74	3240893692/3307266338	Basic_string<...>::length(void) const [7]
		1147.72	14741.81	901806022/989592590	Basic_string<...>::data(void) const [13]

		...	...	...	...
		15430.56	0	989592590/162000611	Basic_string<...>::data(void) const [13]
		51569.68	0	3307266338/162000611	Basic_string<...>::length(void) const [7]
[5]	83.8	69496.80	0	162000611	Basic_string<...>::rep(void) const [5]

		...	...	...	...
		1496.24	50534.74	3240893692/3307266338	Basic_string<...>::compare(basic_string<...> const &, unsigned int, unsigned int) const [4]
[7]	64.1	1526.88	51569.68	3307266338	Basic_string<...>::length(void) const [7]
		51569.68	0	3307266338/162000611	Basic_string<...>::rep(void) const [5]

		2316.60	23254.22	713924/1427848	MainSimulator::loadLinks(Coupled &, Ini &) [9]
		2316.60	23254.22	713924/1427848	Coupled::addInfluence(basic_string<...> const &, basic_string<...> const &, basic_string<...> const &, basic_string<...> const &) [11]
[10]	61.7	4633.20	46508.43	1427848	ProcessorAdmin::processor(basic_string<...> const &) [10]
		1817.76	43273.98	287282416/450903011	Basic_string<...>::compare(basic_string<...> const &, unsigned int, unsigned int) const [4]
		...	...	...	...

		0.12	25768.79	356962/356962	MainSimulator::loadLinks(Coupled &, Ini &) [9]
[11]	31.1	0.12	25768.79	356962	Coupled::addInfluence(basic_string<...> const &, basic_string<...> const &, basic_string<...> const &, basic_string<...> const &) [11]
		2316.60	23254.22	713924/1427848	ProcessorAdmin::processor(basic_string<...> const &) [10]
		0.35	194.65	713924/713934	Model::port(basic_string<...> const &) [54]
		2.97	0.01	356962/356962	Port::addInfluence(Port const &) [86]

Due to the *dynamic loading* of the model in CD++ (i.e., the model is loaded in runtime), CD++ spends most of the time looking and comparing the new and incoming symbols – model names – in a linear fashion. This explains the excessive time spent comparing and handling them. When the simulation is executed, the simulator creates a temporary 'file'

where a virtual model is created based on the model file. To create the model, symbol parsing is used to identify the names of the components of the model and to check if these symbols are repeated in different sections of the model file. Therefore, most of the workload is spent in the parsing of the model, which is dependant on library functions that are borrowed from the compiler; therefore, they are specific to the compiler used.

To check the impact of the symbol parsing libraries used in the overall performance, and to measure the additional time taken by CD++ at initialization for dynamic loading of the model file, a test that involves having CD++ making use of newer version of libraries of GCC to compare its performance with more updated symbol-lookup algorithms.

We followed the same methodology for the HI and HO models, with the same parameters for both of them, for the new HOmod model the initialization test is not practical; for depths of more than 9 levels the OOM Linux service forces a termination of the process because of memory overflow, something similar happens with widths of more than 10.

The comparison of initialization time between CD++ and ADEVS for HI and HO models is given in figures 29 and 30.

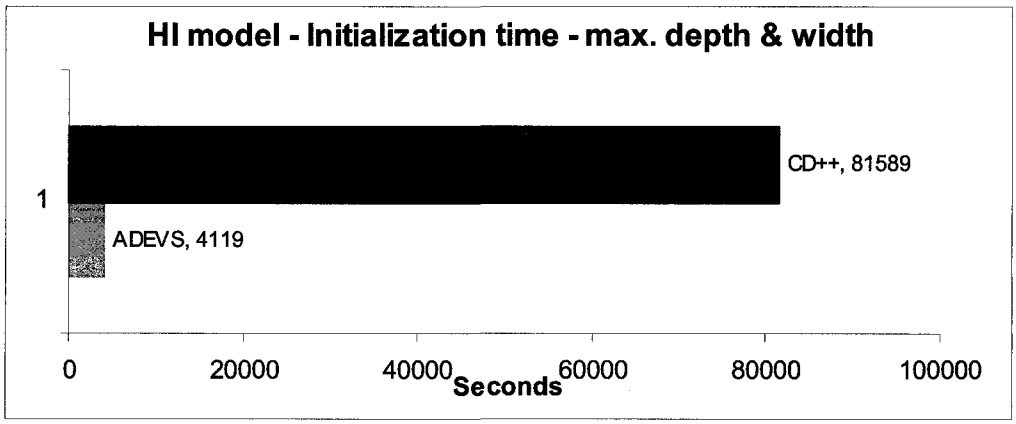

Figure 29: DEVStone Initialization time HI model

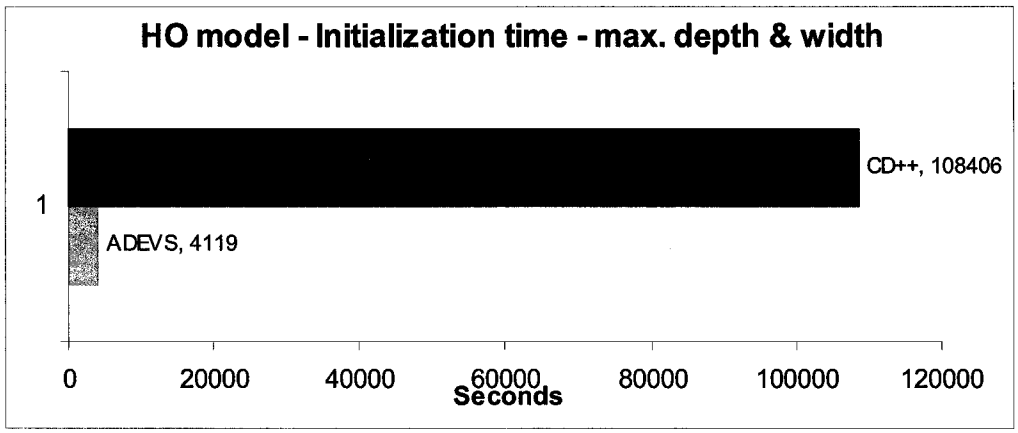

Figure 30: DEVStone Initialization time HO model

For both of them the Call Graph showed similar results as the LI model, having most of the running time looking-up and comparing new symbols with the ones already stored in memory, with similar percentages in the time spent in the look-up and comparison of new symbols from the model file.

In both cases, the links are much more complicated than in the LI model. For the HI model the time spent in the processor block is 26.41 %, and in the function that compares strings the simulator spent 18.85% of the simulation time. In the case of the HO type, the simulator spent 28.70 % in the processor block and 26.51% comparing string arrays.

The main cause for this behaviour of CD++ compared with ADEVS is due to the way that CD++ loads the model. This dynamical loading uses symbol-parsing, look-up and compare which are major setbacks for the overall performance of CD++, on the other hand, the dynamic loading of libraries offers an exceptional flexibility when designing or correcting models for a wide-range of simulations. The dynamic loading of models are heavily dependant on the libraries for symbol parsing borrowed by CD++ from GCC version 2.95.3 (which is required to compile CD++ due to the fact of incompatibilities with the Standard Template Library functions [64] in newer versions of GCC). Considering that the version

2.95.3 of the GCC compiler is fairly old compared with the current version, GCC 4.2.0 as of May of 2007, we should be able to see better performance with a CD++ simulator compiled with the newer version. Nevertheless, CD++ will still present slower simulation time compared with ADEVS, which provides tighter integration between the model and the simulator engine because the parsing of the model-simulator is done at compilation time.

Because ADEVS takes much more time compiling large depth models, another good piece of information would be to compare the performance of both simulators with a model of maximum depth and minimum width. Based on the original test setup, we followed suite and created an event file that provides 10 external events evenly spaced every 0.250 (s) the results of the first simulation with such file are shown in Figure 31.

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
195	2	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

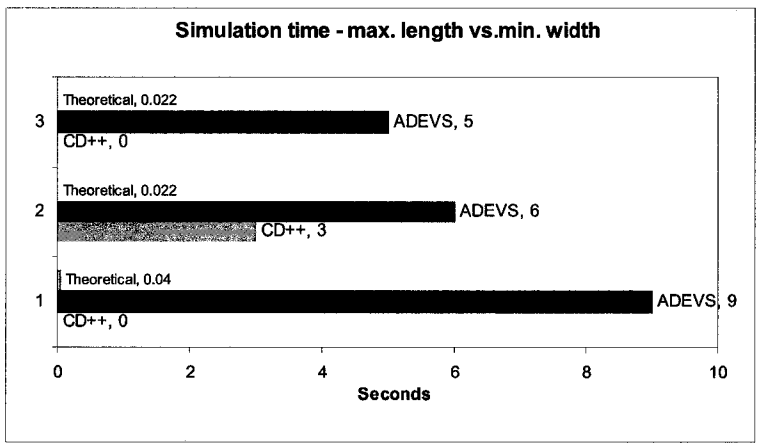

Figure 31: Minimum width and maximum depth of models

In this case, CD++ proves to be faster than ADEVS, mainly because for CD++ the models are loaded in memory as they are needed, regardless of their nature and processed accordingly by the CD++ entity that controls each node, atomic models or coupled models. One interesting note in this case is that for the first scenario where the internal transition function equals the external transition function ADEVS takes almost 50% more time, than the rest of the tests, to finish the simulation, this suggests that ADEVS loads the entire code of the model-simulator to memory, then performs the simulation and finally flushes all the program from RAM and terminates, the reading and writing of the whole file to and from the hard-drive would explain the increase in simulation time. In addition, a check to the memory usage in a second run demonstrated that while ADEVS is running, it takes up to 99% of the available memory right from the beginning while CD++ increases the memory usage incrementally. However, a detailed analysis of the memory usage was not deemed necessary for our purposes because we assume that all the resources will be given to the simulation during its execution, i.e. a dedicated system will be put in place for a simulation and it will only run simulations of a particular model.

Having provided a methodology to analyze the performance of CD++, is now possible to show the flexibility of DEVStone in both different environments and at the same time extract a more reliable tendency of the performance of the simulators. With this idea a set of simulations were executed with different models of similar values for every model, except for the last HOmod model. For the rest of the tests, the simulators were compiled without any option that might slow down the performance. We tested the performance of the simulators based on four main parameters:

- the model type,
- variations in the width of the model,
- variations in the length of the model.
- variations in the real-time running internal and external transition functions.

A total of 11 models where tested, in six of them we varied the width of the models keeping the rest of the data constant, to asses the difference in the internal and external transitions we simulated the same model with equal time spent in the transitions, the internal transition longer than the external transition and finally the external transition longer than the internal transition.

By changing the width of the model, we can focus our analysis on the time that the simulator spends sending messages back and forth to atomic blocks within each level.

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
10	100-600	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

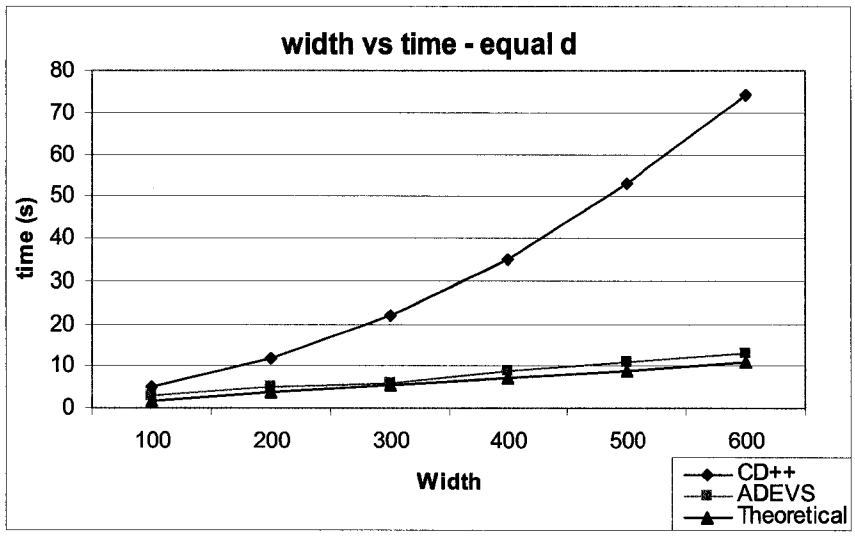

Figure 32: LI Plot for $\delta_{int} = \delta_{ext}$

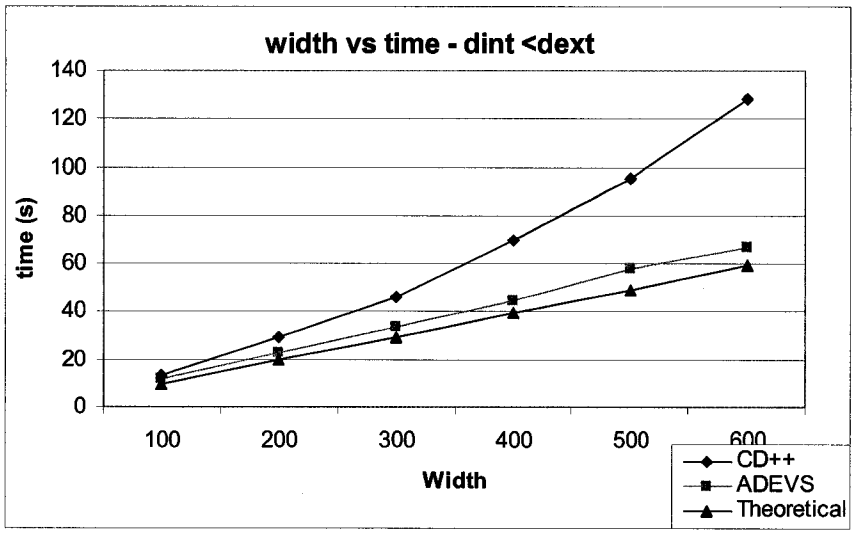

Figure 33: LI Plot for $\delta_{int} < \delta_{ext}$

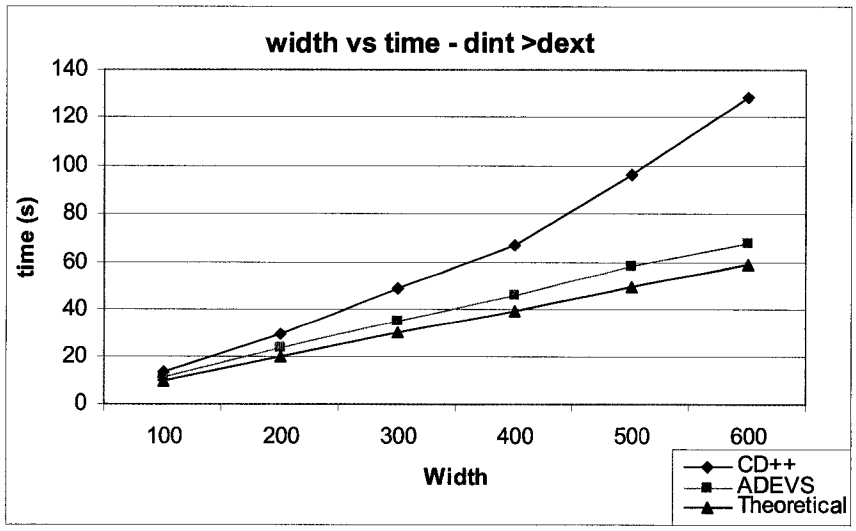

Figure 34: LI Plot for $\delta_{int} > \delta_{ext}$

In Figures 32, 33, 34 it can seen that ADEVS clearly outperforms CD++ by a significant margin for large models, although both simulators are well above the theoretical values, the major difference between CD++ and ADEVS becomes wider when the internal and external transition times are equal; with CD++ exceeding the theoretical result by slightly more than 2 to 6 times the predicted value and ADEVS exceeding the predicted value by just a few seconds difference. One probable cause for this is that CD++ runs using a temporal file where intermediate results are stored and read when necessary whereas ADEVS runs the simulation without any temporary files and depends only in memory availability.

With a similar approach, we could see if this trend remains constant when running the simulation varying the depth variable, in this case by increasing the depth levels of the model we are analyzing the variation in the performance when the interchange messages travel among coupled blocks. The same parameters as in the previous test were used.

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
5-10	100	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

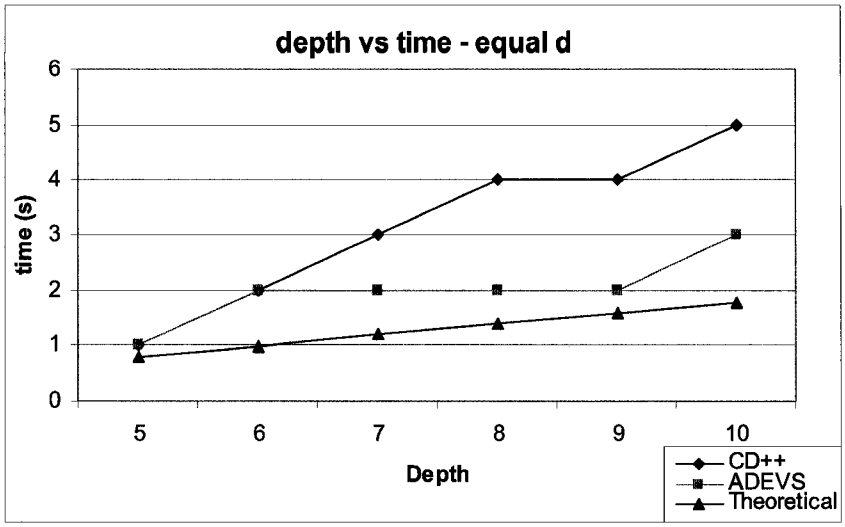

Figure 35: LI Plot for $\delta_{int} = \delta_{ext}$

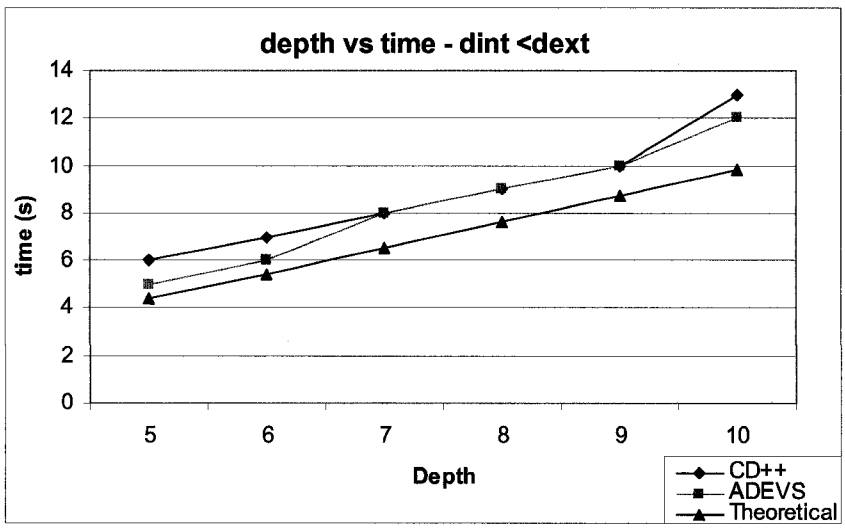

Figure 36: LI Plot for $\delta_{int} < \delta_{ext}$

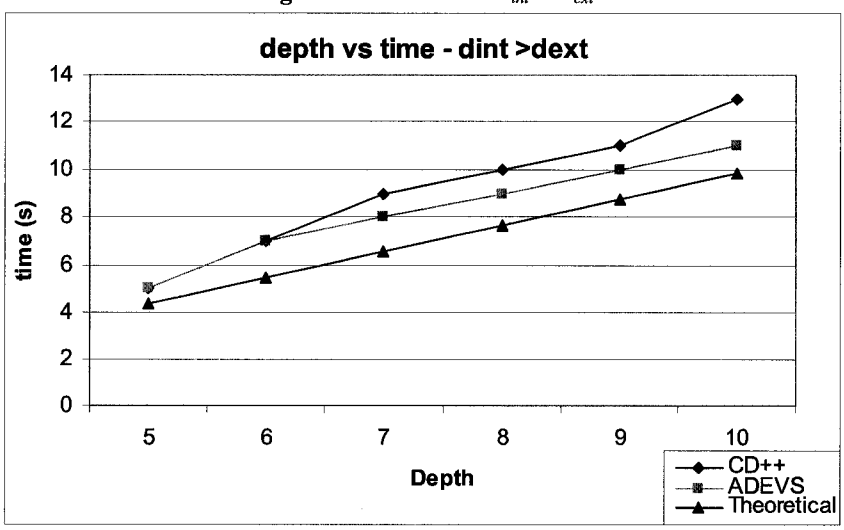

Figure 37: LI Plot for $\delta_{int} > \delta_{ext}$

In Figures 35, 36 and 37 there is a change in the performance. In this particular case CD++ and ADEVS perform practically the same; and in some cases CD++ matches the performance of ADEVS when the internal transition function is less than the external transition function.

HI and HO models share the same equation but with a different structure. The HI type of model connects the atomic blocks in linear fashion but with a greater number of interconnections between the components within the coupled model.

As stated for the simulation runs of HI, we are using the same values used for the depth and width of the LI models. By following the same methodology as before, we start by varying the width of the models and maintaining the depth constant.

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
10	100-600	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

In Figures 38, 39 and 40 we can see that the performance of the simulators have changed dramatically. Although CD++ still lags behind ADEVS when the transition functions are equal, the performance when the internal transition function is less than the external transition is not as big as the one we could expect based on the results of the simulation of LI models. Even more unexpected is the result when the external transition is greater, where for large models CD++ outperforms ADEVS, although the difference in simulation time between simulators can be neglected.

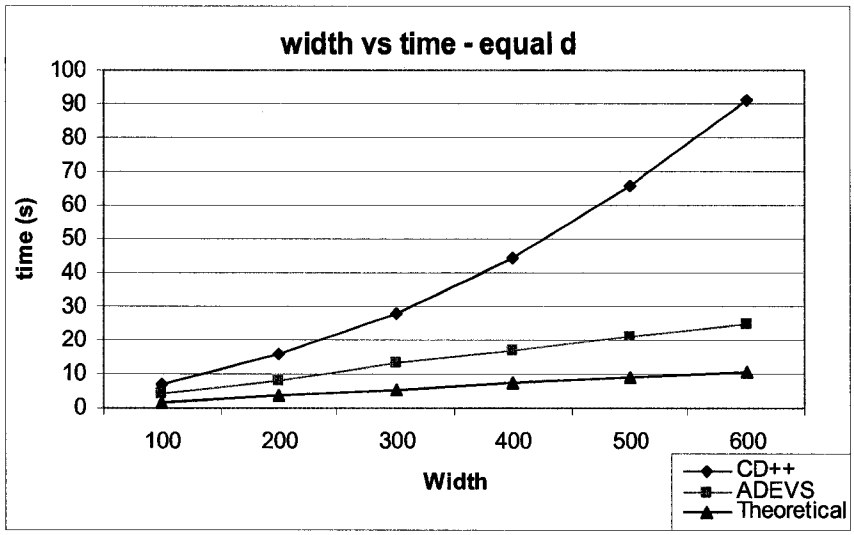

Figure 38: HI Plot for $\delta_{int} = \delta_{ext}$

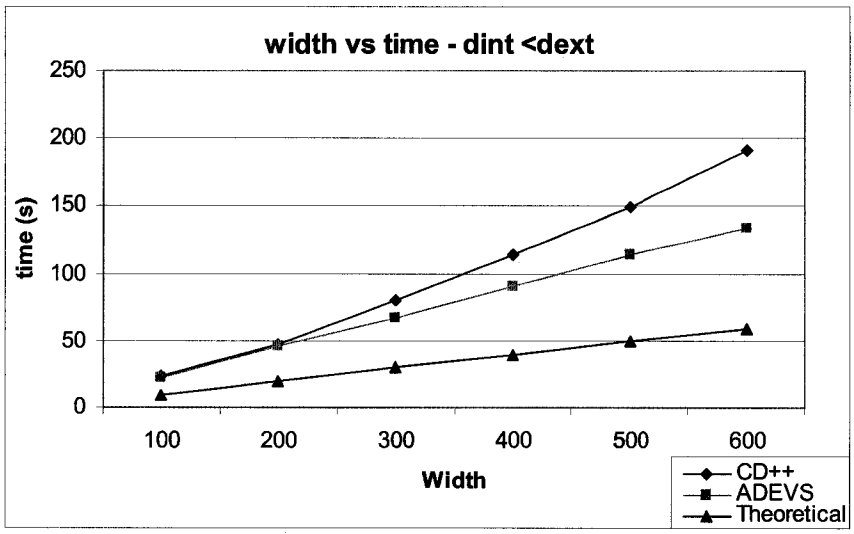

Figure 39: HI Plot for $\delta_{int} < \delta_{ext}$

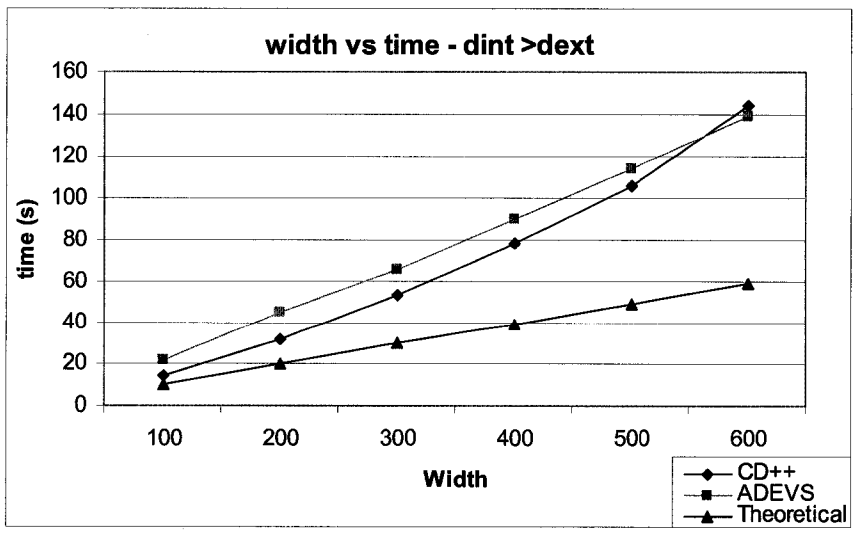

Figure 40: HI Plot for $\delta_{int} > \delta_{ext}$

The same experiment was done by varying the depth of the levels and keeping the width constant. The same three different sets of internal and external transition functions were used. For variable depth HI models with constant width the simulation parameters used were:

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
5-10	100	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

In figures 41, 42 and 43 we cans see the results of varying the depth in the HI model. When the transition functions are equal, ADEVS behaves somewhat better than CD++ as we saw before.

From Figure 41 we can see that both simulators behave in similar fashion when the internal transition is less than the external transition, even though both are well above the theoretical performance of the simulation. In addition, whenever the internal transition is greater than the external transition CD++ surpasses the performance of ADEVS by an ample margin, close to the theoretical values expected for the model.

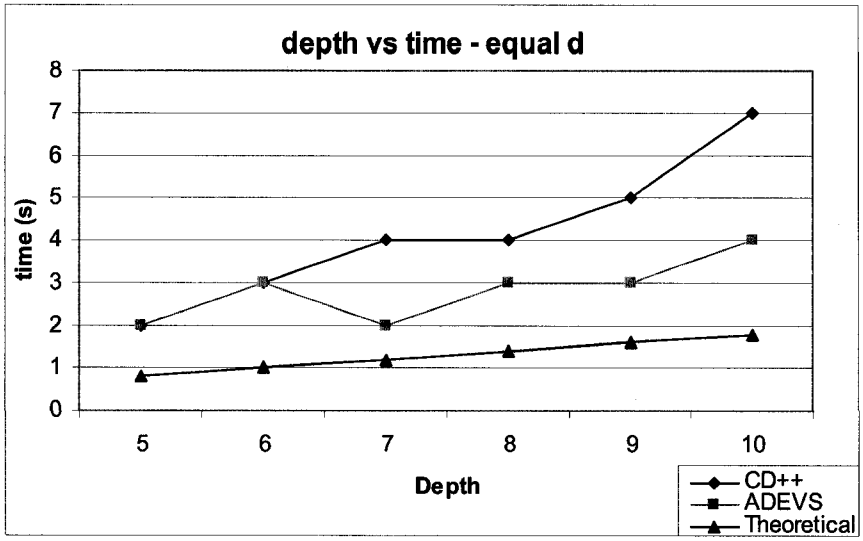

Figure 41: HI Plot for $\delta_{int} = \delta_{ext}$

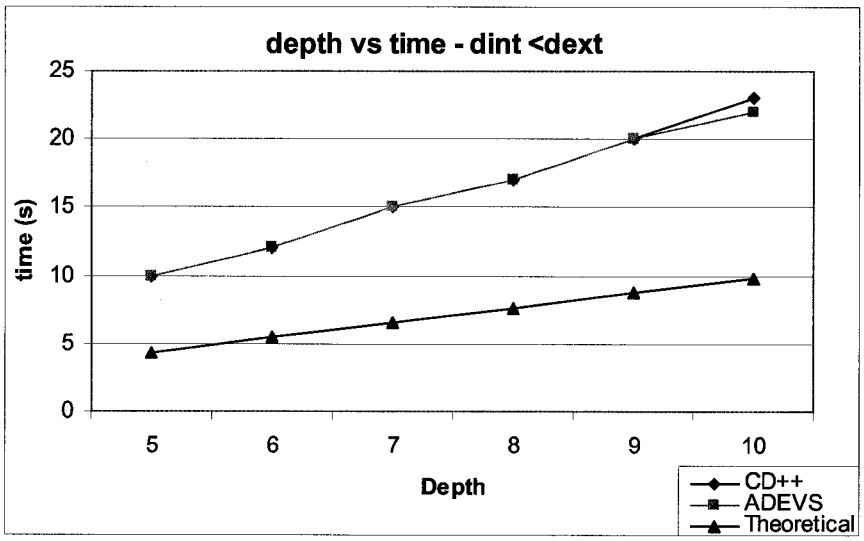

Figure 42: HI Plot for $\delta_{int} < \delta_{ext}$

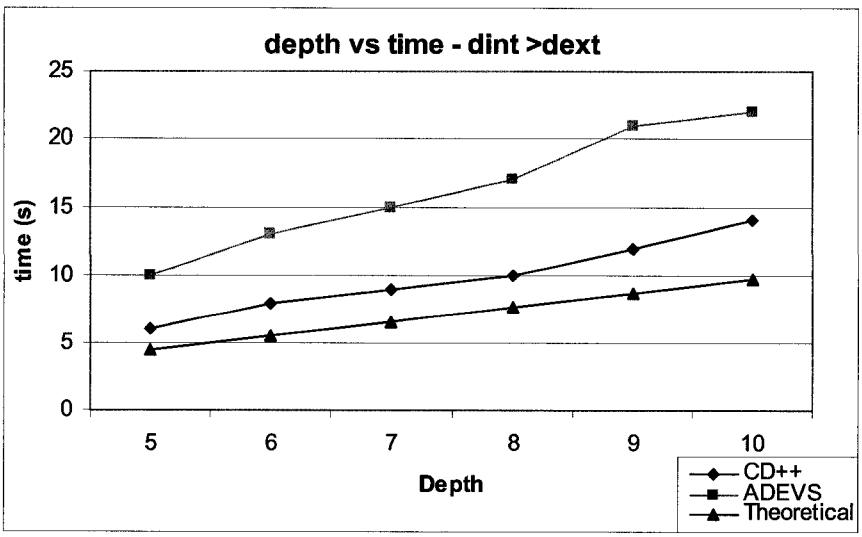

Figure 43: HI Plot for $\delta_{int} > \delta_{ext}$

In HO models, the model structure is similar to HI models but there are more messages coming through the coupled models, via the second input. Following with the test, we choose the same values for the width and depth and run the simulations. The graphs in the next page show the results of such simulations.

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
10	100-600	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

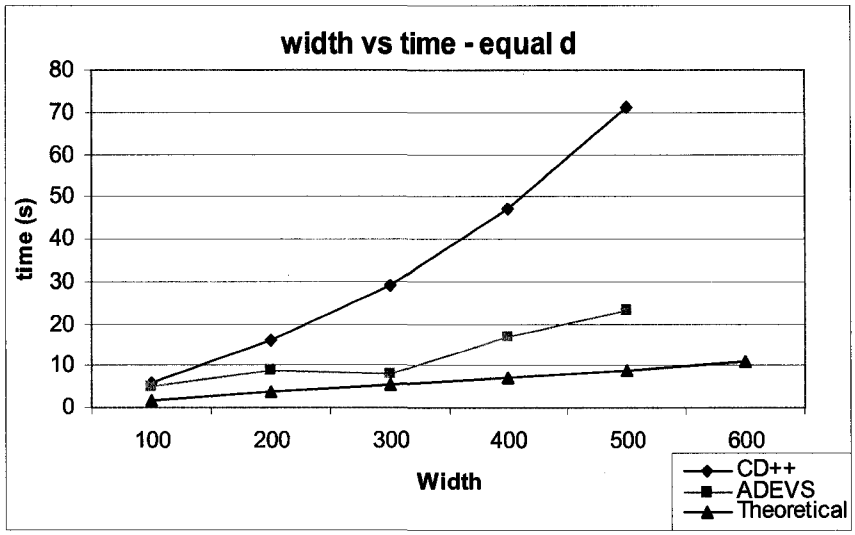

Figure 44: HO Plot for $\delta_{int} = \delta_{ext}$

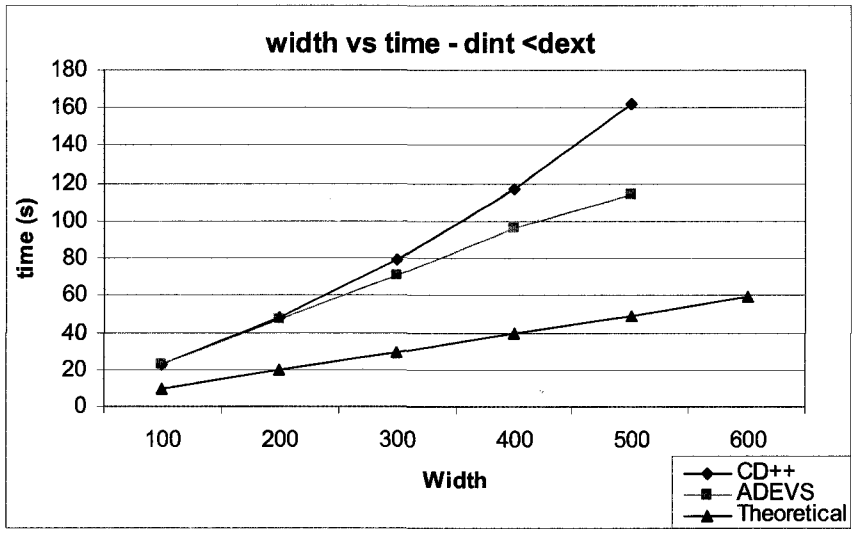

Figure 45: HO Plot for $\delta_{int} < \delta_{ext}$

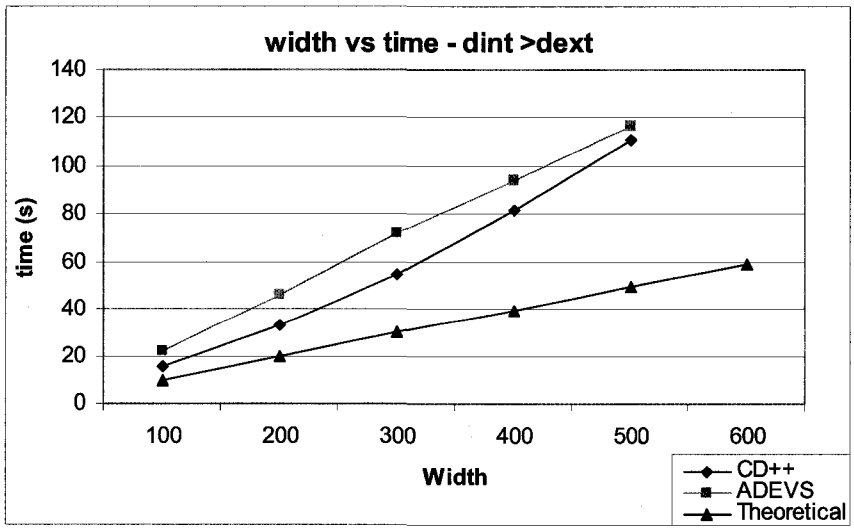

Figure 46: HO Plot for $\delta_{int} > \delta_{ext}$

In the last graphs (Figures 44 - 46), neither simulator could complete the last test for 600 atomic components; both were terminated by the OOM Linux service. In the first graph, with equal transition functions, we can see a drastic difference in the performance of the simulators; CD++ is clearly being outperformed by ADEVS by as much as 4 times the theoretical value, just like the simulation results for the LI models.

When the external transition is greater than the internal transition, CD++ presents similar timing results for relatively small and medium models, but the time increases exponentially for larger models. Moreover, in the last case where $\delta_{int} > \delta_{ext}$ CD++ surpasses ADEVS by a relative small difference but the gap grows thinner along the curve when the width increases.

We can analyze if these last results are repeated when the depth varies and keeping the rest of the values constant:

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
5-10	100	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

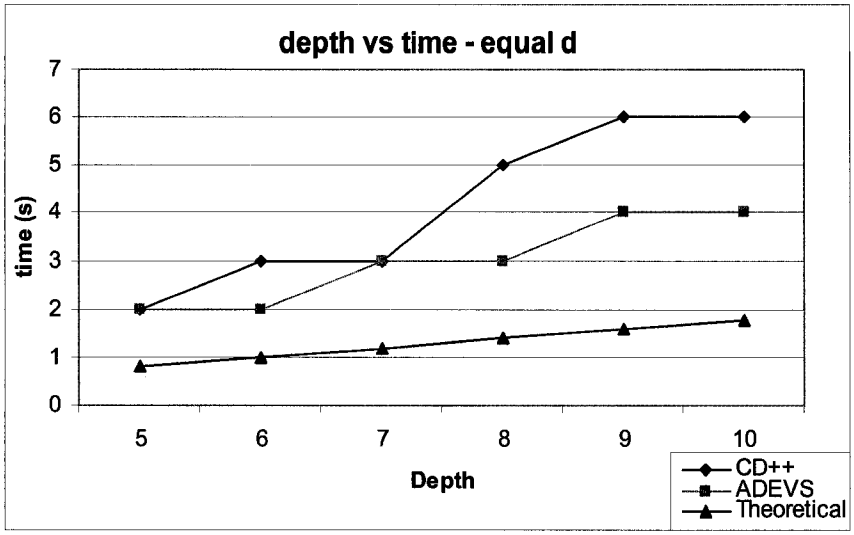

Figure 47: HO Plot for $\delta_{int} = \delta_{ext}$

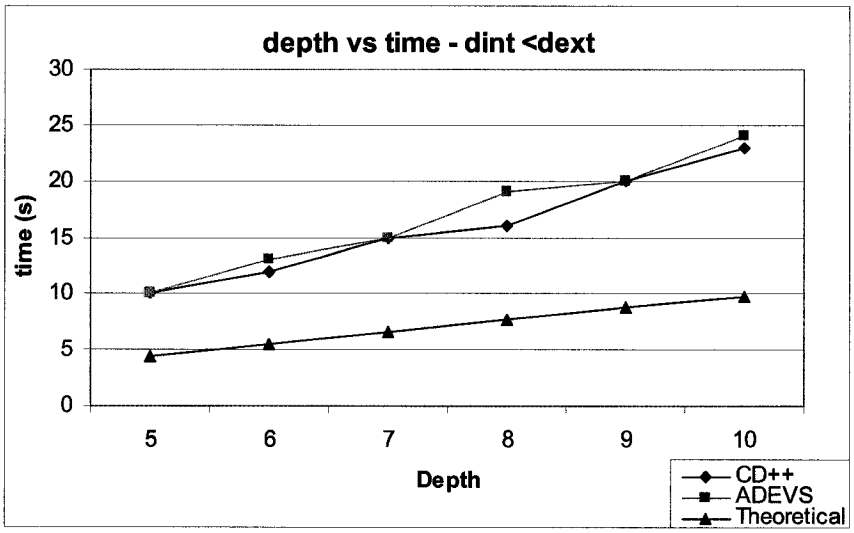

Figure 48: HO Plot for $\delta_{int} < \delta_{ext}$

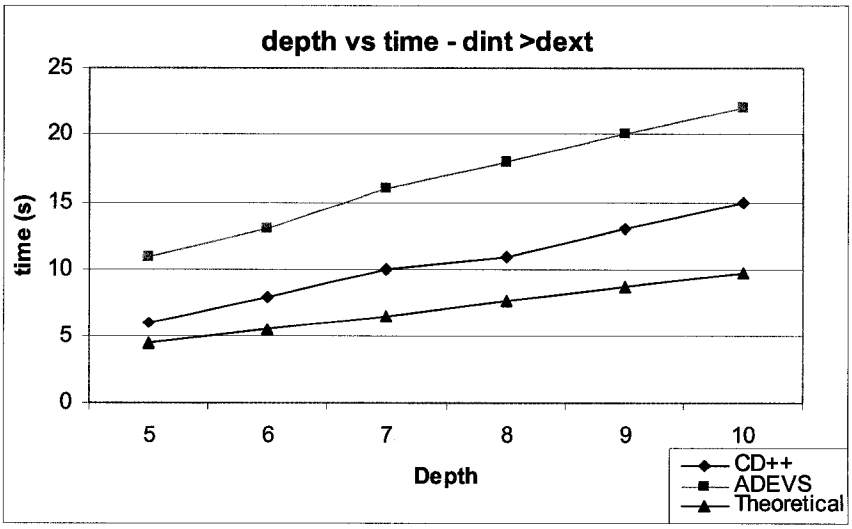

Figure 49: HO Plot for $\delta_{int} > \delta_{ext}$

From the graphs (Figures 47 – 49) it can be seen that the general overall tendency is kept, when $\delta_{int} = \delta_{ext}$ ADEVS behaves better than CD++, and when $\delta_{int} < \delta_{ext}$ both simulators have similar behaviour or very little difference in the performance and lastly when $\delta_{int} > \delta_{ext}$ CD++ outperforms ADEVS by far.

For HOmod models the parameters need to be changed, due to the exponential growth of messages between coupled components, however the time given to each transition function is kept equal. But even then, CD++ had some problems with the last simulation, being terminated by the OOM Linux service. In this case the number of messages passing between components is many times greater than the number of components. The simulation parameters used for the width test were:

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
5	5-8	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

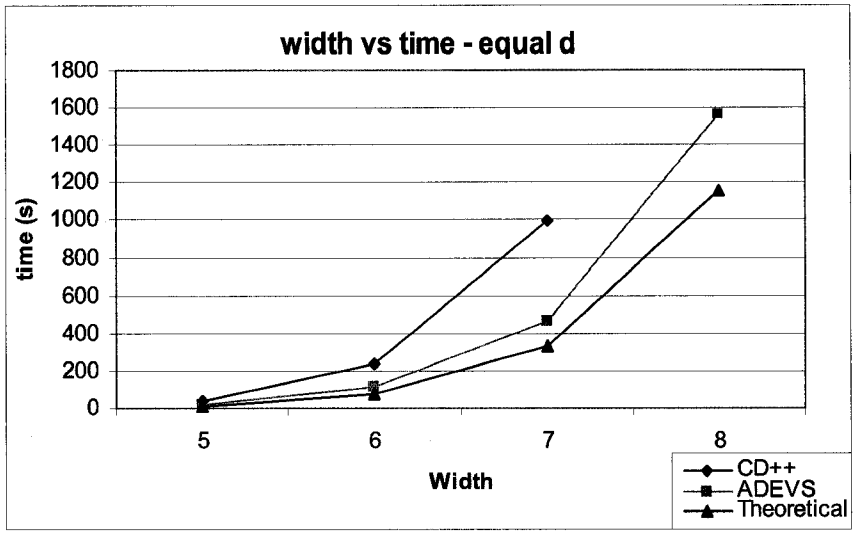

Figure 50: HOmod Plot for $\delta_{int} = \delta_{ext}$

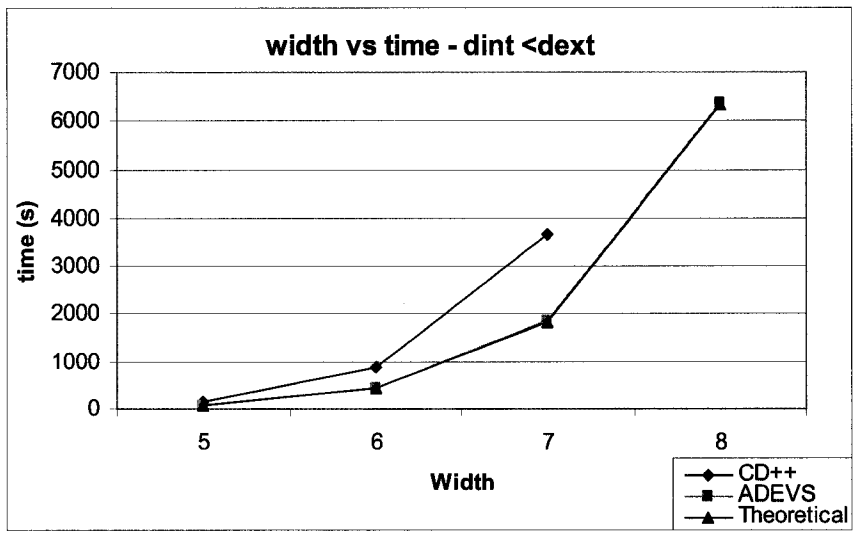

Figure 51: HOmod Plot for $\delta_{int} < \delta_{ext}$

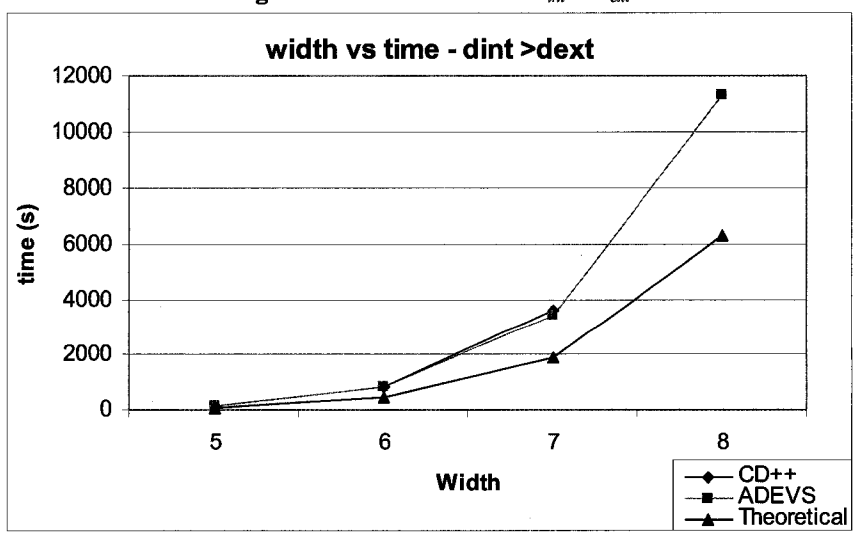

Figure 52: HOmod Plot for $\delta_{int} > \delta_{ext}$

From the graphs (Figures 50 – 52), it can be seen that the general overall tendency is kept constant with some differences: when $\delta_{int} = \delta_{ext}$ ADEVS behaves better than CD++. It is important to note that CD++ could not provide a result for w=8 and was terminated by the OOM service, most likely by the use of symbol look-up library of CD++.

In the second case where $\delta_{int} < \delta_{ext}$ ADEVS and CD++ perform differently from the trend established so far. Furthermore, CD++ was terminated for w=8 by Linux. Nevertheless, the behaviour captured so far suggests that ADEVS performs very close to the theoretical curve, whereas CD++ is almost 80% slower than ADEVS, which is a different behaviour compared to the one seen using previous models of DEVStone.

For the last case, (Figures 53 – 55) both simulators have similar behaviour or very little difference in the performance. When $\delta_{int} > \delta_{ext}$, although inconclusive, CD++ and ADEVS seem to match each other performance for larger values of w.

Following the test scenarios the test was repeated varying the depth this time, the parameters of the simulation were:

Simulation Parameters

		1		2		3	
Depth	Width	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}	δ_{int}	δ_{ext}
3-6	5	0.1ms	0.1ms	0.1ms	1ms	1ms	0.1ms

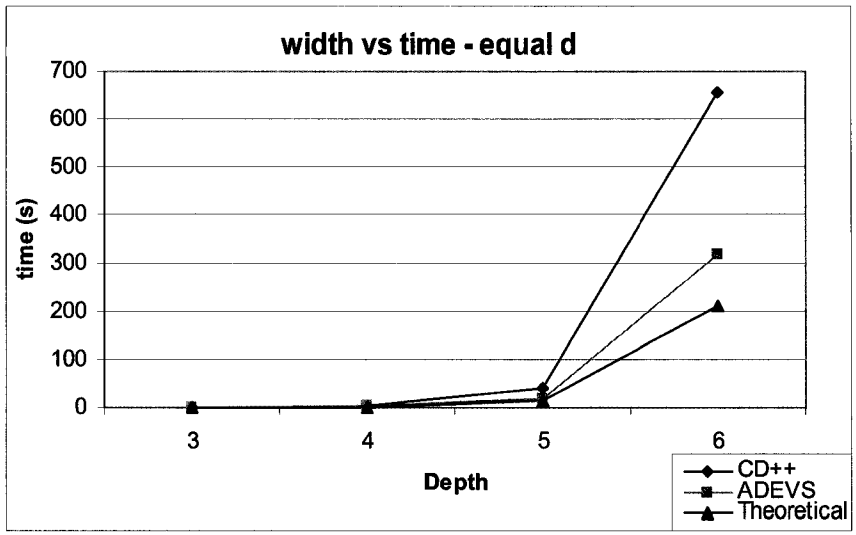

Figure 53: HOmod Plot for $\delta_{int} = \delta_{ext}$

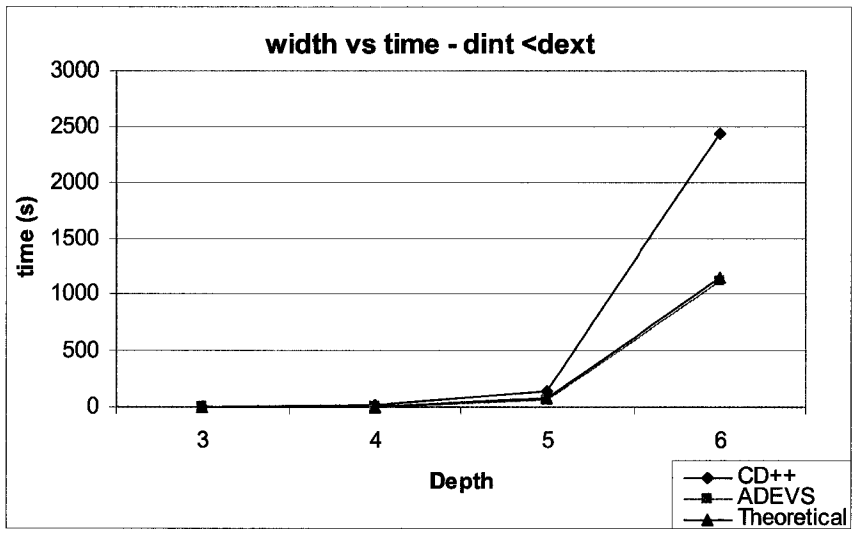

Figure 54: HOmod Plot for $\delta_{int} < \delta_{ext}$

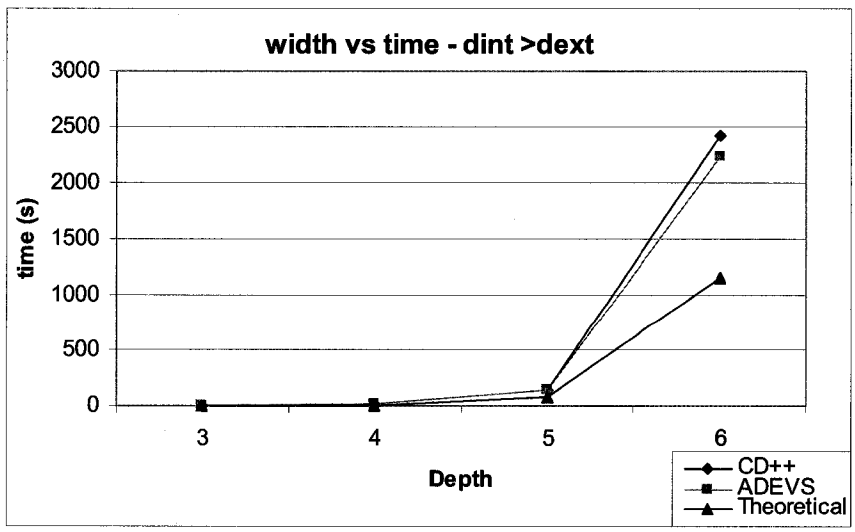

Figure 55: HOmod Plot for $\delta_{int} > \delta_{ext}$

This last test confirms the results provided by the test analyzed before. Where CD++ and ADEVS present a very close performance for models where the internal transition function is greater than the external transition function. For the rest of the cases ADEVS performs better than CD++, although depending on the model and complexity of it. Along with this trend, CD++ offers equivalent performance for models that consume equal time in the transition functions as well as models that have bigger loads in the external transition function.

7. Conclusions and comments

Two components have been added to the CD++ family of tools, a modified benchmarking tool and an extension for the CD++ Builder IDE for Embedded systems based on the ECD++ version of CD++, therefore separate conclusions for each component are given.

The *DEVStone Benchmark* provides a common metric to compare the results that were obtained using different simulation tools; it also enables the analysis of the efficiency of successive versions of the same simulator, such as upgrades or fixes. We used the CD++ simulator and ADEVS to show how to apply the proposed benchmark for comparison purposes, by using this method we have discovered that CD++ has a performance problem related to one particular library in charge of the lookup and comparison of symbols in the dynamic loading of the model, such library is included with the compiler and used by CD++. Although we restricted our case study to CD++ and ADEVS simulation engines, DEVStone may be ported to other DEVS-based simulators.

Using DEVStone, we showed that hierarchical simulation techniques are capable to simulate different types of models, from low message passing to message intensive models with enormous overhead, as in the case of the new model that was developed to increase the passing of messages between coupled and atomic models in DEVStone. This new model will allow the user to test the performance of simple models that require considerable quantities of inter-messages among the components. We also demonstrated that it is possible to classify simulators by their performance, for example CD++ shows better performance than ADEVS when it comes to models that deal with intensive code in the internal transition function, or that ADEVS takes advantage of the tight integration of the model and the simulator at run time, although the price is paid at compilation time; it

also offers less flexibility compared to CD++ when it comes to model development and modification.

On the other hand, by using the new Integrate Development Environment extension designed for ECD++ to model, simulate and implement an embedded system, we demonstrated a main advantage in the development process. Even though the development process involved the modifications of simulator core files to give ECD++ power over the standard PC parallel port, the design was straightforward; using the IDE can improve the development process. The implementation of the RoboCart test case took less than 3 days, instead of at least 2 to 3 weeks that would have been spent in dealing with different intricacies of the simulator and the system. The test case also demonstrated the flexibility of Embedded CD++ toolkit when interacting with real events by the use of simple sensors and actuators; therefore, opening the possibility of creating more complex models and discrete control structures based on the DEVS formalism. The new extension of ECD++ presented in this thesis can use the parallel port for simulation, emulation, testing and validation of embedded system. Utilizing this integrated M&S tool helps maintain consistency among the different phases of design of embedded projects with ECD++ before putting them to real test scenarios where the interaction is with non-deterministic real inputs and outputs.

Undoubtedly more work needs to be done with the real time extension of the Parallel port of ECD++, for the RoboCart an ultrasonic sensor would provide a better solution but the communication's middleware needs to be developed to achieve this. The use of a breadboard for the driver makes the whole system prone to failures making it necessary to integrate the driver into a dedicated printed circuit board.

8. References

1. Schludermann, Harald; Kirchmair, Thomas; Vorderwinkler, Markus. "Soft-Commissioning: Hardware-In-the-Loop-Based Verification of Controller Software". Winter Simulation Conference 2000: Proceedings of the 32nd conference on Winter simulation. Orlando, FL, USA. Society for Computer Simulation International. 2000.
2. Puttré, Michael. "Simulation-based design puts the virtual world to work". DesignNews. February 16, 1998
3. Hu, Xiaolin. "A Simulation-Based Software Development Methodology for Distributed Real-Time Systems". PhD Thesis Dissertation. Department of Electrical and Computer Engineering – University of Arizona.
4. Liu, Jane W.S. Real-time Systems. Prentice Hall. Upper Saddle River, NJ. 2000.
5. Jerraya, A.A.; Wolf, W. "Hardware/software interface co-design for embedded systems". Computer Volume 38, Issue 2, Feb. 2005 Page(s):63 - 69. IEEE Computer Society.
6. Zeigler, B.; Kim, T.; Praehofer, H. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press. 2000.
7. Morse, J.; Hargrave, S. "The increasing importance of software". Electronic Design, Vol. 44 (1), Jan. 1996.
8. P. Paulin, M. Cornero, C. Liem, F. Nacabal, C. Donawa, S. Sutarwala, T. May and C. Valderrama, "Trends in embedded systems technology: An Industrial Perspective". NATO ASI on Hardware/software co-design, Lake Como, Italy, 1995.
9. P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Coossens, "Embedded software in real-time signal processing systems: Application and architectural trends". Proc. of IEEE, vol. 85(3), Mar. 1997, pp. 419-435

10. Paul Robertson, Robert Laddaga, and Howie Shrobe, "Introduction: The First International Workshop on Self-Adaptive Software", Lecture Notes in Computer Science, Volume 1936/2001, pp. 1-10, Springer Berlin, 2001.
11. Hu, Xiaolin. Ziegler, B. "Model Continuity in the Design of Dynamic Distributed Real-Time Systems". IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART A: SYSTEMS AND HUMANS, VOL. 35, NO. 6, NOVEMBER 2005.
12. Huang, D. and Sarjoughian, H. "Software and Simulation Modeling for Real-Time Software-Intensive Systems". Proceedings of the Eighth IEEE international Symposium on Distributed Simulation and Real-Time Applications (Ds-Rt'04) - Volume 00 (October 21 - 23, 2004). DS-RT. IEEE Computer Society, Washington, DC, 196-203.
13. Rastofer, U.; Bellosa, F., "Component-based software engineering for distributed embedded real-time systems". Software, IEE Proceedings, Volume: 148 Issue: 3, June 2001.
14. Wainer, G.; Glinsky, E. "Model-Based Development of Embedded Systems with RT-CD++". RTAS 2004. IEEE Real-Time and Embedded Technology and Applications Symposium May 25-28, 2004. Toronto, Canada.
15. Zeigler, B.; Moon, Y.; Kim, D.; Ball, G. "The DEVS Environment for High-Performance Modeling and Simulation" IEEE Computational Science and Engineering, vol. 4 (3), pp. 61 -71. 1997.
16. Glinsky, E.; Wainer, G.; "DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments". Distributed Simulation and Real-Time Applications, 2005. DS-RT 2005 Proceedings. Ninth IEEE International Symposium on 10-12 Oct. 2005 Page(s):265 – 272.
17. Wainer, G. "CD++: a toolkit to develop DEVS models". Software - Practice and Experience. vol. 32, pp. 1261-1306. 2002.

18. Nutaro, J. ADEVS Internet Homepage. Available on http://www.ornl.gov/~1qn/adevs/index.html. Accessed on Oct. 12, 2006.
19. Troccoli, A.; Wainer, G. "Implementing Parallel Cell-DEVS". Proceedings of 36th IEEE/SCS Annual Simulation Symposium. Orlando, USA. 2003.
20. Glinsky, E.; Wainer, G. "Definition of Real-Time simulation in the CD++ toolkit". Proceedings of the SCS Summer Computer Simulation Conference. San Diego, USA. 2002.
21. Zeigler, B.P.; H.S. Sarjoughian, "Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA". Simulators Interoperability Workshop, 99S-SIW-066.
22. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Framework and Rules. IEEE Std. 1516-2000. September 2000.
23. Sarjoughian, H.S.; Zeigler, B.P. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment". Proceedings of the SCS International Conference on Web-Based Modeling and Simulation, vol. 5, pp. 29-36. San Diego, USA. 1998.
24. Zeigler, B.P.; Kim, J. "Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control". IEEE Transactions on Robotics and Automation. 1993.
25. Kim, T.G. "DEVSim++: C++ based Simulation with Hierarchical Modular DEVS Models". User's Manual CORE Lab, EE Dept, KAIST, Taejon, Korea. 1994.
26. Dávila, J.; Uzcágegui, M. "GALATEA: A multi-agent, simulation platform". Proceedings of the International Conference on Modeling, Simulation and Neural Networks. Mérida, Venezuela. 2000.
27. Filippi, J-B.; Bernardi, F.; Delhom, M. "The JDEVS environmental modeling and simulation environment" Proceedings of the the IEMSS'02 Conference on Integrated Assessment and Decision Support. Lugano, Switzerland. 2002.

28. de Lara, J.; Vangheluwe, H. "ATOM3: A Tool for Multi-Formalism Modeling and Meta-Modeling". European Joint Conferences on Theory And Practice of Software. Grenoble, France 2002.

29. Praehofer, H.; Sametinger, J.; Stritzinger, A. "Discrete Event Simulation using the JavaBeans Component Model". Proceedings of International Conference On Web-Based Modeling & Simulation. California. 1999.

30. Barr, Michael. Embedded Systems Glossary – Netrino Technical Library. http://www.netrino.com/Publications/Glossary/index.php. Accessed on 2007-04-18.

31. Moore, G. "Cramming more components onto integrated circuits". Electronics Magazine 1965. 19 April 1965.

32. Thoen, Filip; Catthoor, Francky. "Modeling, Verification, and Exploration of task-level concurrency in real-time embedded systems". Kluwer Academic Publishers, 2000, pp.46.

33. K. Ghosh, B. Mukherjee, K. Schwan, "A Survey of Real-Time Operating Systems", Technical report, Atlanta, Georgia 30332-0280, College of Computing, Georgia Institute of Technology,1994.

34. Liu, C. L.; Layland, James W. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". Journal of the ACM, Vol. 20, Nr. 1, pgs. 46-61, 1973.

35. Zhao, Wei; Ramamritham, Krithi; Stankovic, J. A. "Preemptive scheduling under time and resource constraints". IEEE Transactions on Computers, C-36(8):949-960, August 1987.

36. Stewart, D. B.; Khosla, P. K. "Real time scheduling of sensor based control systems". In Eighth IEEE Workshop on Real-Time Operating Systems and Software, May 1991.

37. Dertouzos, Michael L.; Mok, Aloysius K. "Multiprocessor on line scheduling of hard real time tasks". IEEE Transactions on Software Engineering, 15(12):1497-1506, December 1989.

38. Sgroi, M.; Lavagno, L.; Sangiovanni-Vincentelli, A., "Formal models for embedded system design", Design & Test of Computers, IEEE , Volume: 17 Issue: 2 , April-June 2000.

39. Hu, X.; Zeigler, B. P. "An Integrated Modeling and Simulation Methodology for Intelligent Systems Design and Testing". Performance Metrics for Intelligent Systems Workshop, August, 2002.

40. Washington, Chris; "HIL simulation boosts automotive design efficiency". Automotive DesignLine, Accessed on May 09, 2007. http://www.planetanalog.com/showArticle?articleID=199501368

41. Ramesh, B.; Jarke, M. "Toward reference models for requirements traceability". Software Engineering, IEEE Transactions on , Volume: 27, Issue: 1 , Jan. 2001.

42. Antoniol, G.; Caprile, B.; Potrich, A.; Tonella, P. "Design-code traceability for object-oriented systems". Annals of Software Engineering vol. 9: 35-58 (2000).

43. Boyd, Joanne L.; Karam, Gerald M. "Designing reactive systems for strong traceability". International Workshop on Software Specifications & Design. Proceedings of the 7th international workshop on Software specification and design Carleton University, 1993.

44. Janka, R. S.; Wills, L. M.; Baumstark, L. B. "Virtual Benchmarking and Model Continuity in Prototyping Embedded Multiprocessor Signal Processing Systems", IEEE Transactions on Software Engineering, Vol. 28, No. 9, September 2002.

45. Li, Lidan; Wainer, Gabriel; Pearce, Trevor. "Hardware In The Loop Simulation Using Real-Time CD++". Department of Systems and Computer Engineering. Carleton University. Accessed on 2007-05-02.

46. Del Bianco, Vieri; Lavazza, Luigi; Mauri, Marco; et al. "Towards UML-based formal specifications of component based real-time software" pp. 118 - 134 Lecture Notes in Computer Science Publisher: Springer-Verlag Heidelberg Volume: Volume 2621 / 2003.

47. Kim, K.H., "Object Structures for Real-Time Systems and Simulators". IEEE Computer, August 1997, pp.62-70.

48. See [16].

49. Wainer, Gabriel; Yu, Henry. "ECD++: An Engine for Executing DEVS Models in Embedded Platforms". Summer Computer Simulation Conference 2007 (SCSC 2007). San Diego, California (USA), July 15-18, 2007.

50. Joshua J. Yi, Lieven Eeckhout, David J. Lilja, Brad Calder, Lizy K. John, James E. Smith, "The Future of Simulation: A Field of Dreams". Computer, vol. 39, No. 11, pp. 22-29, Nov., 2006. IEEE Computer Society.

51. Glinsky, E.; Wainer, G. "Performance Analysis of Real-Time DEVS Models". Proceedings of the SCS Winter Simulation Conference. San Diego, CA. 2002.

52. Troccoli, A.; Wainer, G. "Performance Analysis of Cellular Models with Parallel Cell-DEVS". Proceedings of the SCS Summer Computer Simulation Conference. Florida. 2001.

53. Chiari, F.; Delhom, M.; Filippi, J-B.; Santucci, J-F. "A GIS based methodology for the modeling and the simulation of watersheds". Proceedings of the ATW 2000 Conference. Corsica, France. 2000.

54. Kim, K.; Kang, W. "CORBA-Based, Multi-Threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One". International Conference on Computational Science and its Applications. Assisi, Italy, 2004.

55. Zeigler, B.; Moon, Y.; Kim, D. "DEVS-C++: A High Performance Modeling and Simulation Environment". 29th Hawaii International Conference on System

Sciences (HICSS'96) Volume 1: Software Technology and Architecture. Hawaii, USA. 1996.

56. ECLIPSE. Eclipse 3.1 Online Manual. www.eclipse.org.
57. OpenSSH. "OpenSSh Manual Pages". http://www.openssh.org/manual.html. Accessed on May 02, 2007.
58. SecuriTeam. "Multiple Vendors Telnet Vulnerability". http://www.securiteam.com/unixfocus/5RP0D2K4UQ.html. Accessed on January 10, 2007.
59. Corbet, Jonathan; Kroah-Hartman, Greg; Rubini, Alessandro. "Linux Device Drivers, 3rd Edition". O'Reilly. February 2005.
60. Department of Systems and Computer Engineering, Carleton University. Ottawa, Canada.
61. Weicker, R. P. "Dhrystone: A synthetic systems programming benchmark". Communications of the ACM, volume 27, pages 1013-1030, 1984.
62. Wainer, G. et al. "CD++ A tool for DEVS and Cell-DEVS Modelling and Simulation. User's Guide". Draft. August 2004.
63. Felanson, Jay; Stallman, Richard. "GNU gprof - The GNU Profiler User's Guide". Free Software Foundation. 1998.
64. Chidisiuc, C.; Wainer, G. "CD++Builder: a toolkit to develop DEVS models". In *Proceedings of DEVS Symposium 2007*. Norfolk, VA. 2007.
65. http://www.sce.carleton.ca/faculty/wainer/papers/ISC03RTDEVS.pdf
66. Li, L.; Pearce, T.; Wainer, G. "Interfacing Real-Time DEVS models with a DSP platform". Proc. Of Industrial Simulation Symposium. Valencia, Spain. 2003.
67. STMicroelectronics. "L293B Push-Pull Four Channel Driver". Datasheet.
68. Texas Instruments. "SN74LS04 Hex Inverter". Datasheet.
69. Harries, Ian. "Interfacing to the IBM-PC Parallel Printer Port". http://www.doc.ic.ac.uk/~ih/doc/par/. Accessed on April - 10 - 2007.

70. Martin, Fred G. "Robotic Explorations, A Hands-On Introduction to Engineering". Prentice Hall, Upper Saddle River 2001.