E-CD++: AN ENGINE FOR EXECUTING DEVS MODELS IN EMBEDDED ENVIRONMENTS

By

Yinfeng Henry Yu

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfilment for the degree of

Master of Applied Science in Systems and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

Canada

ACKNOWLEGEMENTS

I would like to thank the invaluable supervision and support from Dr. Gabriel Wainer during the development of this work. Working with him has been a challenging and yet rewarding experience.

I would also like to thank Carleton University, especially the department of Systems and Computer Engineering, for providing the excellent research opportunities and facilities.

This thesis is dedicated to my wife, Rui. Without her enduring support and prayers, this work may not be able to complete.

Abstract

The DEVS (Discrete Event System Specification) formalism defines a formal Modelling and Simulation (M&S) framework for Discrete Event Dynamic Systems (DEDS). The RT-DEVS formalism is a real-time extension to DEVS. RT-DEVS is adequate to model Real-time Embedded Systems (RTES). This work introduces the Embedded CD++ (E-CD++) toolkit. It provides an execution engine that can run DEVS models in embedded environments. E-CD++ supports the RT-DEVS formalism, and can be used as a development tool for RTES. The target system can be first studied and modelled entirely in DEVS. The DEVS models are then executed by E-CD++ on embedded platforms where they can interact with the real-world events in real time. In E-CD++ execution environment, the DEVS models can also interact with real hardware surrogates. When a DEVS component is fully tested in the embedded environment, it can be replaced by its physical counterpart, and this step can be repeated until all the components are replaced by their target counterparts. This development approach enables instantaneous transition from modelling to implementation. We also made more efforts than just implementing RT-DEVS in order to make E-CD++ an adequate real-time execution engine. We improved E-CD++ performance by deploying the Flattened Coordinator Technique. We also implemented the GGAD Graphical Modelling tool, so that the modeller can define DEVS models using graphical notations. Lastly, to illustrate with real applications, we used E-CD++ to built an Automated Manufacturing System (AMS), which is a real-time application consisting of microprocessors and mechanical devices.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ASIC	Application Specific Integrated Circuits
AM	Atomic Model
AMS	Automated Manufacturing System
API	Application Programming Interface
CM	Coupled Model
CVDS	Continuous Variable Dynamic Systems
DEDS	Discrete Event Dynamic Systems
DEVS	Discrete Event System Specification
E-CD++	Embedded CD++
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GGAD	Generic Graphical Advanced environment for DEVS
GUI	Graphical User Interface
M&S	Modelling and Simulation
MSDE	Modelling and Simulation-Driven Engineering
P-DEVS	Parallel DEVS
RT-DEVS	Real-time DEVS
RTES	Real-time Embedded System
RTS	Real-time System
SBC	Single Board Computer
UML	Universal Modelling Language
UML-RT	UML for Real-Time

Chapter 1 Introduction

Real-time systems (RTS) can be characterized as those whose correctness of operation depends not only upon the logical correctness but also upon the time at which it is performed. Current state-of-the-art RTS are typically **embedded systems**, which are advanced computer applications consisting of hardware, software, and various mechanical and electrical devices. Examples are like nuclear power stations, Automated Manufacturing Systems and car airbags. These **Real-Time Embedded Systems (RTES)** typically deliver data from/to devices interacting with the surrounding environment within deadlines ranging at millisecond scales.

Due to unique characteristics of RTES, the RTES design needs to face several special challenges that need not to be dealt with by that of other systems:

- The design needs to meet the timeliness requirements. RTES must provide correct outputs to external events or inputs within a time limit. A RTES can be categorized as either a soft or a hard real-time system, depending on the strictness of its timeliness requirements. For hard real-time, the design must meet the timeliness requirements with zero tolerance on delay, while, for soft real-time, limited tolerance can be allowed for very small delays.

- The design needs to meet the constraints on resources requirements (e.g., limited memory and processing power). Many RTES may also have constraints on power consumption, because they are deployed in environments where grid-electricity is not commonly available (e.g., inside mobile phones or remote devices).

- The design needs to deal with the hardware/software partition problem. The embedded system design space is formed by combinations of hardware and software components, which is also referred to as *hardware/software codesign*. The design decision on dividing the target system into hardware and software

components is referred to as *hardware/software partitioning*. The hardware/software partition problem is NP-complete [KS03], which is why an optimal design for a RTES can be very hard to achieve.

- The design needs to cope with the target systems' increasing scalability requirements. With the advance of the manufacturing technologies, more and more hardware components (e.g., ASICs and FPGAs) are integrated to form a single RTES. Furthermore, RTES are making use of networking technologies to exchange information or inter-work among each other. Networking makes it possible for hundreds of devices working together to complete larger tasks. Consequently, scalability becomes an important design issue.

- The design needs to cope with the target systems' increasing complexity. With the rapid deployment of cheaper and more power microprocessors, RTES are capable of supporting more and more complex applications.

We find that, due to the challenges listed above, no adequate and robust design framework exists today that is capable of carrying out optimal design solutions to RTES. Our study show that the deficiencies of the existing development methods of RTES mainly come from two weak areas: the development lifecycle and the system verification. The deficiencies in the development cycle could be attributed to the fact that no unified methodology or design framework exists today that can be adequately applied throughout the entire design cycle. Some tools/methods are better in one development stage, while others are better in other stages. Consequently, different tools and methods are used in different development stages, resulting in inconsistencies among analysis, design, test, and implementation. Consequently, when the development tasks switch towards the target environment, the early models are often abandoned [WG02]. For example, in the analysis stage, MATLAB may be used to build mathematical models to analyze data and algorithms. However, these mathematical models are rarely used at the design stage, where UML (Unified Modelling Language) is a more commonly used tool. However, for the implementation phase, UML models are inadequate comparing with programming

languages, such as C, because UML models cannot be directly executed on the target systems with real-time performance.

Another area of deficiencies is system verification. Since the current state-of-the-art RTES are complex systems that consist of a mix of software embedded in and interacting with hardware components and that also need to respond to real-world events in real time, correctness of RTES design is very difficult to achieve. Although formal methods for RTES design are promising, they have difficulties in scaling up when the complexity of the system increases. **Modelling and Simulation (M&S)** techniques, instead, are adequate for testing particular conditions, regardless of the application's size. However, no M&S technique exists today that can provide the same degree of adequacy to study RTES as that provided by mathematical methods to study continuous variable systems. The lack of adequate formal modelling methods makes RTES development become an ad-hoc process that is expensive, time consuming and error prone. For example, current methods for software construction for RTES require a difficult and expensive testing effort with no guarantee for a bug-free product. [LG05] listed three current approaches to RTES testing, with none of them being adequate:

- Formal specifications. When applying formal specifications, the requirements of the System of Interest are formally defined, and formal methods are subsequently applied to prove correctness. These techniques have had some success, but they are difficult to apply when the complexity of the system scales up.

- M&S techniques. M&S techniques and tools are proved to be to be helpful in designing complex systems. Nevertheless, no practical or automatable approach exists to perform the transition that exists between the modelling and the development phases, and this often results in model artefacts being abandoned, resulting in increased initial costs. Consequently, even though they provide improved products, M&S studies are not carried out, or they are used for analyzing individual subsystems, later discarding the developed software.

Simultaneously, M&S frameworks are not as robust as their formal counterparts are.

- A third kind of technique widely used to ensure that a RTS conforms to a specification is Software Testing, i.e., the execution of the software system with actual values. Although this method cannot guarantee the correctness of the application, it provides a practical solution that tries covering the largest possible number of system use scenarios.

To overcome these problems, the solution is to develop a formal methodology that is adequate to be applied to every design stage throughout the entire development lifecycle. Modelling and Simulation-based Development of RTES relies on simulation based modelling for developments of RTES. We propose a novel Simulation-based Development framework for RTES, based on a formal method called DEVS (Discrete Event Systems specification) [Zei76, Zei00]. To be used as the final target architecture for products, DEVS provides a formal foundation to M&S that has been proven to be successful in different complex systems. We choose DEVS to model RTES because of the following reasons:

- The DEVS formalism is a formal method based on mathematical theories. So, the correctness of DEVS models can be formally validated.

- DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition, which makes it adequate to model RTES.

- DEVS not only proposes a framework for model construction, but also defines an abstract simulation mechanism that is independent of the model itself. This mechanism provides a high level description of how the simulation of DEVS models should be executed. Based on this mechanism, it is possible to develop a real-time embedded execution environment in which DEVS models are run and

interact with the real hardware surrogates, so that the target system can be developed in hardware-in-the-loop.

Our methodology is based on successive prototyping and refinement, which combines the advantages of a simulation-based approach with the rigor of a formal methodology, which is well suited for RTES development. It consists of the following steps: Modelling phase, model verification phase, and Incremental Model replacement phase.

1. *Modelling phase*: The modeler defines the DEVS model for the target system. The modeler may have a choice to define the DEVS models using a high-level graphical notation, which makes it easier to understand system structure and behaviors.

2. *Model Verification phase*: This phase is concerned with the transformation of the model specification into an executable model. The models obtained from the previous phase are used to automatically derive simulation, and experimentation is done in a virtual environment. The simulation runs on high processing power workstations.

3. *Incremental Model Replacement phase*: Once the models are verified in a virtual environment, they are then executed in a real-time environment. The tested components are incrementally replaced by their target counterparts interacting with the actual setting.

4. This cycle is incrementally repeated up to the moment where the system is fully developed and tested.

This new methodology defines a unified design process for RTES. By simulating the models, RTES designers will be able to used formal methods analyze every detail of system status and requirements. Furthermore, tested models will be directly replaced by their real counterparts, so that instantaneous transition from modelling to development can take place.

The proposed simulation-based development framework overcomes the design deficiencies that we found in other design methods. Our methodology covers every aspect of the RTES design and provides a consistent design framework throughout the entire development lifecycle. Early design models will no longer be abandoned at the development phase. Rather, they are directly applied to the implementation, as our approach creates a seamless transition from modelling to development. Furthermore, the automation of the transition from model definition to real-time execution eliminates source level coding and ad-hoc program tailoring, and thus reduces the design efforts.

The proposed methodology also provides a sound mechanism for system verification. Since the DEVS formalism is derived from formal mathematical methods, the DEVS-based system design can be formally validated against the target system's specification. In addition, since the DEVS models are modular, subcomponents of larger models can be validated individually. Moreover, when the validated models are translated to RT executives running in the real world, their behaviours can also be easily verified by comparing the execution results with that obtained in the simulated world. If the two results fail to agree, the simulated solution can be revised in the simulated world for retest. If, on the other hand, a subcomponent is verified in the real world, it can be replaced by real hardware. This technique enables incremental transition from the simulated models to the actual hardware counterparts. As a result, all the subcomponents of the system-under-develop are formally verified when the development is finally finished.

1.1 Contributions

The DEVS real-time execution engine plays an important role in the proposed methodology. The design starts entirely in the simulated world. However, with the help of the execution engine, the simulated models can be executed in the real world environment. This is one of the major differences between our method and the traditional M&S methodology. This work focused on developing the DEVS real-time execution engine.

- We developed an embedded toolkit called **Embedded CD++ (E-CD++)**. E-CD++ integrates the execution of DEVS models with hardware surrogates and allows the simulated models to interact with other real components in a real-time embedded environment.

- The model execution by E-CD++ complies with the **RT-DEVS (Real-Time DEVS)** specification. RT-DEVS is a real-time extension of the DEVS formalism. It provides a sound theoretical foundation for modelling RT systems. Furthermore, it provides a framework for the construction of hierarchical models in a modular manner, allowing for model reuse and reducing development time and testing. It also allows hierarchical decomposition of the model by defining a way to couple existing DEVS models.

- Performance is an important factor for the success of E-CD++, because it must execute the DEVS models in real-time. We devoted lots of efforts to improve the performance of E-CD++. We implemented a technique that simplifies the model hierarchy while preserving the original model relations. By simplifying the model hierarchy, E-CD++ reduces the runtime overhead incurred by the traversal of the hierarchy. We also did the mathematical analysis on performance improvements of this technique.

- Since E-CD++ runs in the real world, we implemented wall-clock time in E-CD++. So, the activities run by E-CD++ are measured against the physical time. This is another difference between E-CD++ and other DEVS simulation toolkits, which use virtual time for simulation.

- E-CD++ also supports DEVS graphical notations; so, DEVS models generated by the graphical modelling tool can be directly executed in E-CD++.

1.2 Thesis Organization

The rest of this work is organized as follows:

Chapter 2 reviews the state-of-the-art in the M&S field. The chapter surveys the existing M&S technologies used in RTES field. It then describes the specifications of DEVS, P-DEVS, and RT-DEVS. It also provides a brief survey on the DEVS-based toolkits existing today. We conclude, based on the review of the state-of-the-art, that no M&S methodologies or toolkits exists today that is adequate to develop RTES formally. We, therefore, develops E-CD++ which servers as the DEVS RT executive in our new methodology.

Chapter 3 discusses the functionalities of E-CD++. The discussion covers four major functionalities: GGAD graphical notation, P-DEVS realization, Flattened Coordinator technique, and finally the realization of Time Interval Function.

Chapter 4 reveals the design and implementation details of E-CD++. It provides a software architecture overview, followed by the detailed descriptions of four major software modules: the Main Simulator, modelling subsystem, simulation subsystem, and messaging subsystem.

Chapter 5 is a case study in which we put all the pieces together to show how E-CD++ is used in the new methodology to develop a real application. The development of an AMS is demonstrated in detail. The case study illustrates step-by-step how the AMS is designed using hardware-in-the-loop. In addition, the experimental results are used to test the E-CD++ functionalities including GGAG graphical modelling, performance improvements by Flattened Coordinator, and P-DEVS' confluent functions.

Finally Chapter 6 states the conclusions of this work and outlines the possible future work.

Chapter 2 M&S for Embedded Systems Design

This chapter explores the state-of-the-art in the use of M&S for embedded systems design. Many different M&S methodologies exist for embedded systems. We will survey on these methodologies and compare their strengths and limitations. From there, we will aim to find the best methodology for the RTES development. We will first provide an exposure to DEVS (Discrete EVent Systems Specification), an M&S formalism that supports hierarchical and modular modelling. We will show both the strengths and the limitations of DEVS. Then we will introduce Parallel DEVS, which is an extension to DEVS, and how it can overcome those DEVS limitations. Thirdly, we will introduce another DEVS extension called Real-time DEVS. It provides a formal modelling framework for real-time systems, making it an ideal choice for RTES development. We will also show how Real-time DEVS can be realized based on Parallel DEVS. Finally, to make use of DEVS models in embedded systems design, DEVS simulators must be developed. At the end of this chapter, therefore, a brief survey on the existing DEVS-based simulation toolkit will be given.

2.1 M&S Methodologies

As the results of the increasing embedded systems design complexity and the shortening of the time-to-market design window, two revolutionary changes have emerged in this field [Ern98]. First, the concurrent design of hardware and software has displaced the traditional sequential design. Further, hardware and software design begins before the system architecture, or even the specification, is finalized. As a result of these changes, M&S have become a very important step in embedded systems design [CEP99].

This section provides a survey on various existing M&S methodologies for modelling embedded systems. The M&S process, in general, begins with defining the constraints imposed on the system under design, for example constraints on cost, performance, and physically dimensions. An **experimental frame** captures these constraints. Within the constraints, the M&S process captures the features of the system under design and

describes its functionality**.** In this step, entities are identified, and an abstract representation, a **model**, is constructed. Once the model is constructed, it needs to be executed. This is done by a **simulator**, which consists of a computer system that executes the model's instructions to generate its behaviour. To complete the cycle, the results obtained are compared to those of the real system for model validation.

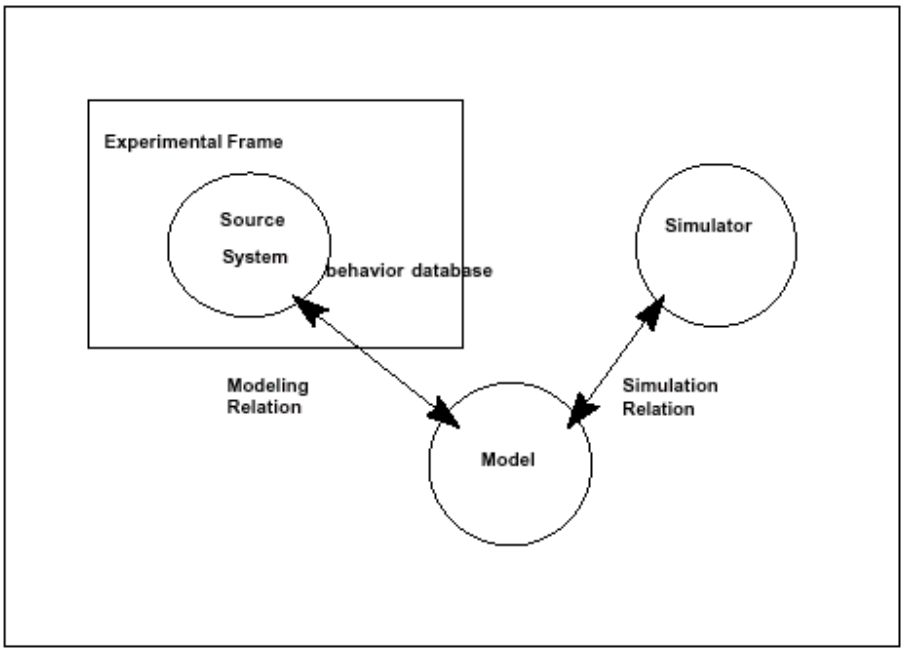

Figure 1 The basic entities and their relationships [Zei00]

The basic entities are linked by two relations [Zei00] (Figure 1):

- *modelling relation.* Links the real system and model, defining how well the model represents the system or entity being modeled. In general terms a model can be considered valid if the data generated by the model agrees with the data produced by the real system in an experimental frame of interest.

- *simulation relation.* Links the model and simulator. It represents how faithfully the simulator is able to carry out the instructions of the model.

Several M&S methodologies have been used for creating embedded systems. A brief description of a non-comprehensive list is given below.

- **Unified Modelling Language (UML)**. UML is a standardized specification language for object modelling. UML is a widely adopted general-purpose modelling language that includes a graphical notation used to create an abstract model of a system, referred to as a **UML model**. UML provides a suite of methods that are well suited for generic software construction. However, while UML is a widely adopted methodology to model software architecture, it is neither adequate nor intended to be used to design hardware components. Therefore, UML may not be suitable for software-hardware codesign [Mar02]. Another drawback of the UML model is the lack of formal proofs of its correctness; so validation and verification efforts become non-trivial, especially for complex UML models.

- **UML for Real-Time (UML-RT).** UML-RT is an extension of UML. It offers additional modelling constructs based on Real-time Object-Oriented Modelling, such as event, action, resource, and schedule. UML-RT is targeted for modelling complex, event-driven, and distributed real-time systems [HS04]. However, UML-RT does not fundamentally solve the limitations inherited from the original UML. First, UML-RT supports concurrency to the same extent as defined by UML. An important shortcoming of this is the inability to guarantee processing of events using priority settings [HS04]. Secondly, UML-RT does not provide formal simulation algorithms – it simply executes the models' logical specifications – which undermines having a well-defined relationship between model specifications and model simulations. Moreover, UML-RT runs on top of the target system's real-time operation system (RTOS). Consequently, the resolution of time and the multi-task schedule are dependent on the underlying RTOS.

- **Finite State Machines (FSM)**. FSM is well known for describing control systems [CEP99]. This model consists of a set of states, a set of inputs, a set of outputs, a function which defines the outputs in terms of inputs and states, and a next-state function. FSM do not allow concurrency of states, nor does it support

hierarchical constructions. Another shortcoming is the exponential growth of the number of states as the system complexity rises. Nevertheless, a number of extensions and variations of FSM have been proposed attempting to overcome the weakness of FSM. **Statecharts**, with its commercially available simulator **StateMate** [Har96], is the most widely adopted FSM extension for modelling embedded systems. Statecharts have the structure of finite-state automata enhanced with three important features: hierarchy, concurrency, and broadcast communication. One of the disadvantages of Statecharts, however, is the lack of formal modelling capability [SER00]. Statecharts employs UML to specify models, as opposed to using formal models. This may make the synthesis process difficult, as synthesis can be applied only if the precise mathematical meaning of a design description is applied [SLS00]. Also, Statecharts do not formally specify explicit timing, which is an important aspect in embedded systems.

- **Dataflow Graphs**. A dataflow graph consists of a set of compute nodes and directed links connecting them representing the flow of data [Alu03]. Dataflow graphs are quite popular in modelling data-dominated systems, such as signal processing [LDNA03]. While dataflow graphs are well capable of modelling dataflow systems, their semantics, however, do not well support event-driven reactive systems [Ern98].

- **MATLAB**. MATLAB is a simulation toolkit (http://www.mathworks.com), which provides a technical computing and data analysis development platform for system designers. It features integrated tools that provide user access to its math, analysis, visualization, and programming capabilities. MATLAB was built upon solid mathematical schematics. It is often used to solve differential equations and/or provide Fourier transformations. Therefore, MATLAB is suitable for modelling **Continuous Variable Dynamic Systems** (**CVDS**), whose behaviours can be best described and studied by differential equations. For instance, [RCL00] used MATLAB to model and simulate Neuro-Fuzzy systems, in which differential equations and Fourier Transformation functions were used

for the modelling. MATLAB has its limitations as well. It cannot adequately model discrete event systems, such as event-driven reactive systems and dataflow systems, where differential equations are inadequate to serve as the modelling schematics. Another limitation is that MATLAB is unable to adequately perform when many objects existed in the model [JRH03], limiting its ability to model large complex systems.

- **Discrete Event Simulation**. In contrast to CVDS, **Discrete Event Dynamic Systems** (**DEDS**) are systems whose variables are discrete and whose time advance is continuous. Simulation mechanisms for DEDS systems assume that changes of state will take place at discrete points of time, upon the occurrence of an event. An **event** is a change of state that occurs at a specific point of time $t_i \in \mathbf{R}$. The occurrences of events are asynchronous. In between event occurrences, states of DEDS are unaffected.

Due to the nature of digital computing, most of the embedded systems are DEDS. Therefore, DEDS simulation is well capable of describing embedded systems. This is the primary reason why our research adopted discrete event simulation as our M&S methodology.

Discrete event simulators are concurrent software that simulate DEDS. Communication between processes in DE simulators is accomplished by **message passing**. A message is an artificial event that occurs in some instance of physical time. Thus, each message has an event's value and is marked with a time stamp. Each process in a DE simulator is executed when it receives a message (i.e., input events) and produces output events (messages) with the same (zero delay) or a larger time tag. The order of execution of multiple processes that have events at the same time is unspecified. Different DE simulators resolve this problem in different manner. Thus, [SLS00] states that the DE model is ambiguous, in case of simultaneous events.

Although using M&S to design RTES has a promising potential [CEP99], this survey shows that none of the surveyed methodologies is perfect for RTES development. However, due to the DEDS nature of the RTES, Discrete Event Simulation seems to be most suitable for RTES. Our focus, therefore, becomes to find the solution to resolve Discrete Event Simulation's limitation on ambiguity, to make it adequate for modelling RTES. That is, we want to find a discrete event modelling formalism that gives the modeller the full control of defining a deterministic behaviour of the model upon the occurrence simultaneous events.

2.2 The Original DEVS formalism

DEVS (**Discrete EVents Systems Specification**) [Zei76, Zei00] is a Discrete Event Simulation formalism for modelling and simulating DEDS systems. In DEVS, a model is specified as a black box with a state and a duration for that state. When the duration time for the state expires, an output event is sent, an internal transition takes place and the model changes its current state. A change of state can also occur when an external event is received. Then, a complete model is defined by describing the set of states a model goes through, the internal and external transition functions, the output function, and the state duration function. DEVS models can be put together by linking the outputs of a model to inputs of other models to form **coupled models**. Models made out of only one component are called **atomic models**.

DEVS not only proposes a framework for model construction, but also defines an **abstract simulation** mechanism that is independent of the model itself. This mechanism is high level description of how the simulation of DEVS models should be executed by a **simulator**. Two kinds of **simulators** are defined, one for atomic and another one for coupled models, this latter known as a **coordinator**. These simulators progress through the simulation by exchanging messages as described by the abstract simulation mechanism.

A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components. An **atomic model (M)** is defined by:

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

where

X	is the *set of external events*;
Y	is the *set of internal events*;
S	is the *set of sequential states*;
$\delta_{ext}: Q \times X \rightarrow S$	is the *external state transition function*; where $Q = \{ (s,e) / s \in S, e \in [0, ta(s)] \}$ and e is the elapsed time since the last state transition.
$\delta_{int}: S \rightarrow S$	is the *internal state transition function*;
$\lambda: S \rightarrow Y$	is the *output function*;
$ta: S \rightarrow R_0^+ \cup \infty$	is the *time advance function*;

A DEVS model is in a state $s \in S$ at any given time. In the absence of external events, it remains in that state for a lifetime defined by $ta(s)$. A transition that occurs due to the consumption of time indicated by $ta(s)$ is called an **internal transition**. When $ta(s)$ time expires, the system outputs the value $\lambda(s)$ and then changes to a new state given by $\delta_{int}(s)$. On the other hand, an **external transition** occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$ where s is the current state, e is the time elapsed since the last transition and $x \in X$ is the external event that has been received.

The **time advance function** can take any real value between 0 and ∞. A state for which $ta(s) = 0$ is called a **transient state**. In contrast, if the $ta(s) = \infty$ then s is said to be a **passive state**, in which the system will remain perpetually unless an external event is received.

The following figure in [Gli04] shows the description of states and variables in DEVS models:

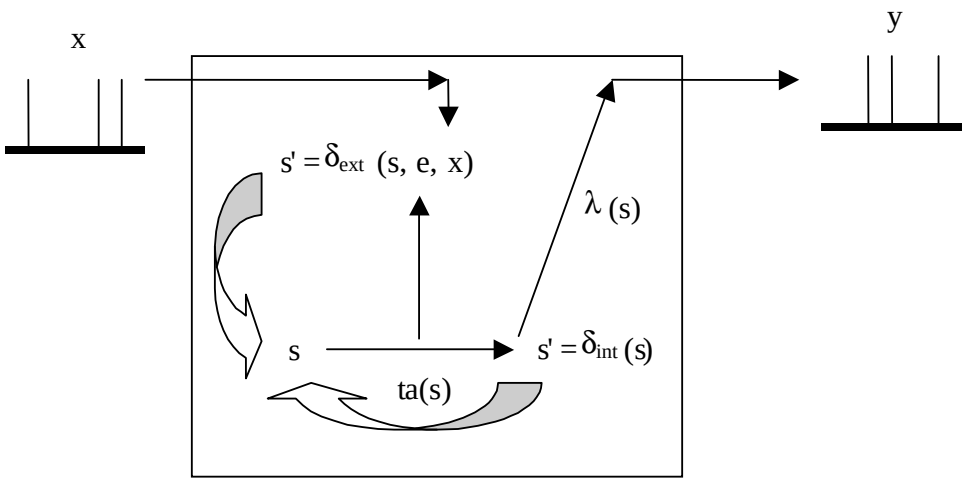

Figure 2 DEVS Semantics [Gli04]

A DEVS **coupled model (CM)** is composed of several atomic or coupled submodels. It is formally defined by:

$$CM = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select \rangle$$

where

D Is a *set of components*;

for each *i* in *D*,

M_i is a *basic DEVS component* (*i.e.* a coupled or atomic model);

for each *i* in $D \cup \{ self \}$,

I_i is the *set of influencees* of *i* (*i.e.* models that can be influenced by outputs of model *i*);

for each *j* in I_i,

$Z_{i,j}$ is the *i-to-j output-input translation* function

Select is the *tie-breaker* function;

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

$M_i = \langle X_i, Y_i, S_i, , \delta_{i\,int}, \delta_{i\,ext}, \lambda_i, ta_i \rangle$ is a DEVS model

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

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

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

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

select: subset of D $\rightarrow$ D

such that for any non-empty subset E, select (E) $\in$ E.

A *coupled model* groups several DEVS into a compound model that can be regarded, due to the closure property, as a new DEVS model. This allows hierarchical model construction.

In addition, each *coupled model* has its own input and output events, as defined by the X_{self} and Y_{self} sets. When external events are received, the coupled model has to redirect the inputs to one or more components. Similarly, when a component produces an output, it may have to map it as an input to another component, or as an output of the coupled model itself. Mapping between ports is defined by the *Z* function.

Two types of ambiguities may rise in DEVS simulations. The ambiguity rises when multiple components are scheduled for internal transitions at the same time in a coupled model. The way the DEVS formalism solves this ambiguity is by the use of the *select* function. The function defines an order over the components so that only one component of the group of imminent models is allowed to have $e = 0$. The other imminent models are divided in two groups: those that receive an external output from this model, and the rest. The former will execute their external transition functions with e = ta(s), and the latter will be imminent during the next simulation cycle which may require again the use of the select function to decide which model will execute first. This tie-breaking approach, however, is a potential source of errors since the serialization produce may not reflect the correct system's behaviour upon the occurrence of simultaneous events.

The second type of the ambiguities may rise in an atomic model when it receives an external event at the exact time when an internal transition is scheduled. It is not clear which transition this model should execute first, for the DEVS formalism does not

specify the order. So, two alternatives exist: to execute the external transition first with $e = ta(s)$ and then the internal transition, or else to execute the internal transition first followed by the external transition with $e = 0$. It is up to the simulation software to decide which alternative to choose. This serialization constraint, however, may again cause errors.

2.3 Parallel DEVS Formalism

While the DEVS formalism suffers from its serialization constraints, the **Parallel DEVS (P-DEVS)** formalism [Cho94a] was introduced to resolve this issue. A **P-DEVS model** is described as a set of basic and coupled models. In addition, the model's interface was also revised. A model will now have input and output ports through which all interaction with the environment takes place. Events determine values appearing on such ports. A model receives outside events through its input ports. Upon reception of such events, the model description must determine how it responds to them. In addition, internal events arising within the model change its state, and manifest themselves as events on the output ports to be transmitted to other model components.

Atomic models are still the most basic constructions, which can be combined with other models into coupled models. A Parallel-DEVS coupled model satisfies the closure property [Cho94b], so it can be seen as another basic model. Therefore, Parallel-DEVS preserves the hierarchical properties of the original DEVS formalism.

The **P-DEVS atomic model** has the following structure:

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

where

$X_M = \{(p,v)|\ p \in IPorts, v \in X_p \}$ is the set of *input ports and value*s;

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

S is the set of *sequential* states;

δ_{ext}*:* $Q \times X_M^b \rightarrow S$ is the *external state transition functio*n;

δ_{int}: $S \rightarrow S$ is the *internal state transition functio*n;

δ_{con}: Q x $X_M^b \rightarrow S$ is the *confluent transition functio*n;

$\lambda : S \rightarrow Y_M^b$ is the *output function;*

$ta : S \rightarrow R_0^+ \cup \infty$ is the *time advance functio*n;

with Q := { (s, e) | s ∈ S , 0 ≤ e ≤ ta(s) } the set of total states.

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

A **P-DEVS coupled model (CM)** is defined by:

$$CM = <X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC>$$

where

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

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

M_d is a set of atomic models, and *D* is a set of the atomic models' names, where

for each $d \in D$

$M_d = (X_d , Y_d , S, \delta_{ext} , \delta_{int}, \delta_{con}, \lambda, ta)$ *is a DEVS basic structure*

with $X_d = \{(p,v) \mid p \in IPorts, v \in X_p \}$;

$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p \}$;

The couplings are subject to the following conditions:

- *external input couplings (EIC)* connect external inputs to component inputs:

$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$

- *external output couplings (EOC)* connect component outputs to external outputs:

$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$

- *internal couplings (IC)* connect component outputs to component inputs:

$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$

No direct feedback loops are allowed, i.e., no output port of a component may be connected to an input port of the same component i.e.,

$((d, op_d), (e, ip_d)) \in IC$ implies $d \neq e$.

- Range inclusion constraints: the values sent from a source port must be within the range of accepted values of a destination port, i.e.,

$\forall((N, ip_N), (d, ip_d)) \in EIC : X_{ipN} \subseteq X_{ipd}$

$\forall ((a, op_a), (N, op_N)) \in EOC : Y_{opa} \subseteq Y_{opN}$

$\forall ((a, op_a), (b, ip_b)) \in IC : Y_{opa} \subseteq X_{ipb}$.

Comparing with DEVS, P-DEVS has the following 2 capabilities that DEVS lacks of:

- Be able to give the modeller a complete control over the collision behaviour when a component receives events at the time of its internal transition via the use confluent function δcon. This function will define a new model's state when there is a collision between internal and external transitions. Basically, this function will allow the modeller to specify how the model should behave in the presence of collisions.

- Be able to eliminate the necessity for tie-breaking simultaneously scheduled events, which is done by the SELECT functions in DEVS. In P-DEVS, the external and output functions no longer handle one event at a time. Instead, bags of events are now being handled, allowing then for simultaneous processing of multiple events. In other words, P-DEVS provides parallel activation of all imminent children of a coupled model.

2.4 Real-time DEVS (RT-DEVS) Formalism

The **RT-DEVS** [HSKP97] is a formalism for real-time discrete event systems modelling. It is an extension of the DEVS formalism that provides a seamless framework for the development of real-time control software that includes modelling, design, analysis, simulation, and implementation. The RT-DEVS has additional specifications that are not in the original DEVS formalism: the time interval function and the weak synchronization communication mechanism [SK05]. Since in real-time systems, an event occurrence time may not be an exact value but an interval, the time advance in RT-DEVS is given by a time interval. An atomic model in RT-DEVS formalism, RTAM, is given by the following seven-tuple [SK05]:

$$RTAM = < X_M , Y_M , S, \delta_{ext} , \delta_{int}, \delta_{con}, \lambda, ti >$$

$$ti : S \rightarrow R_0^+ \text{ x } R_0^+ \text{where } ti(s)|_{min} \leq ta(s) \leq ti(s)|_{max}, s \in S$$

Note that RTAM is the same as the original DEVS atomic model, expect that the time advance function is replaced by the time interval function *ti*.

The definition of coupled models defined by RT-DEVS is the same as that defined by P-DEVS.

The DEVS formalism does not explicitly define a communication mechanism between components coupled together. The RT-DEVS, by contrast, explicitly defines the communication mechanism, called *weak synchronization*. It has two characteristics:

- Concurrency on simultaneously scheduled events. For example, suppose there are two real-time atomic models A and B, and A's output port is connected to B's input port. As a result, A's output function generates an external event to B. In RT-DEVS, A's internal transition and B's external transition takes place concurrently.
- Synchronization loss: A real-time component that is trying to make an internal transition should not block other components that are not ready for synchronization. For example, suppose that atomic models A and B have no

connections via any ports. Then A's internal transition changes its state alone. That is, B's state remains unchanged, causing synchronization loss.

In practice, the implementation of P-DEVS provides weak synchronization, because it provides concurrent activation of all imminent children of a coupled model, and also because it also guarantees synchronization loss [Cho94a]. Therefore, the RT-DEVS formalism can be constructed by implementing both the time interval function and P-DEVS.

2.5 Applying DEVS to Embedded Systems Design

DEVS technology has been usually applied to large-scale dynamic systems, with implementations running on workstations and servers. As these systems focus on the high level modelling and simulation, another branch of DEVS application is on real-time event-based control [HZC01]. These low level applications exist largely on embedded systems, which are usually characterized as "intelligent devices" consisting of computer hardware and real-time software. This work mainly studies how to use DEVS technology to design real-time embedded systems.

Comparing with serial DEVS, P-DEVS is a major advance in modelling real-time systems, because it provides an appropriate basis to develop simulation models exhibiting concurrent behaviour. However, while P-DEVS provides sound modelling principles to characterize structural and behaviour aspects of real-time systems, recent research suggests that transforming (or mapping) DEVS models to actual designs of real-time embedded systems is non-trivial [HS04].

Recent research, therefore, has been focusing on developing schemes to support the transformation from simulation modelling to designs of real systems. One attempt was the DEVS-on-a-chip approach, which implements DEVS on a microprocessor that has limited memory and processing ability [HZC01]. It creates a just-as-needed real time environment. This effort, however, did not implement a full scale of RT-DEVS

specifications on the chip. As a result, it only demonstrates the capability of creating real-time embedded systems that have relatively simple compositions.

Another research effort in this area focused on how to use RT-DEVS as a framework to develop hardware-in-the-loop applications [LPW03]. These applications are complex as a result of the high degree of interaction between software and hardware components. Therefore, the development of these applications is a challenging process in which M&S can become essential. The technique of applying RT-DEVS to develop hardware-in-the-loop applications seamlessly integrates simulation models with hardware components and also enables incremental transition from the simulated models to the actual hardware counterparts.

2.6 DEVS-based simulation toolkits

Prior to this work, many DEVS-based toolkits have already been developed by different research groups. A brief survey on these tools has been given by [Gli04] as follows:

- ADEVS [Nut06] supports the construction of discrete event models based on a variant of the P-DEVS formalism. It includes support for dynamic structure models based on the Dynamic DEVS formalism [Uhr01a].
- DEVS-C++ [Zei06] is a high performance simulation environment that allows portability of models across platforms at a high level of abstraction. It uses a set of C++ classes, called containers, to implement serial and parallel simulation.
- DEVS-Scheme [Zei93] is a knowledge-based environment implemented in Scheme for discrete-event model construction and simulation. It allows combining symbolic and hierarchical, modular discrete-event modelling approaches.
- DEVS/HLA [Zei99b] is an HLA-compliant M&S environment implemented in C++ that supports high level model construction. It greatly simplifies the underlying programming details required to establish and participate in an HLA federation.

- DEVS/Grid [Seo04] is an M&S framework implemented using Java and Globus toolkit for Grid computing infrastructure.
- DEVSCluster [Kim04] is a CORBA-based, multi-threaded distributed simulator implemented in Visual C++. It transforms a hierarchical DEVS model into a non-hierarchical one to ease the synchronization of the distributed simulation.
- DEVSJAVA [Sar98] is a DEVS-based Simulator that supports high-level modelling.
- GALATEA [Dav00] is offered as a family of languages to model multi-agent systems to be simulated in a DEVS multi-agent platform.
- JDEVS [Fil02] is an M&S environment that enables discrete-event, general purpose, object-oriented, component-based, GIS, (Geographic Information System) connected, collaborative, visual simulation model development and execution.
- JAMES [Uhr01b] is a Java-based simulation environment that allows the modeler to describe agents and their environment as situated automata.
- PyDEVS is a simulator developed in ATOM3 [Del02], a tool for multi-paradigm modelling. DEVS models are constructed using the ATOM3-DEVS tool, which generates Python code to be executed with the PyDEVS simulator.
- PowerDEVS [Kof03] is an M&S toolkit developed in C++ for hybrid systems. Atomic DEVS models can be graphically coupled in hierarchical block diagrams to create complex systems.
- SimBeans [Pra99] is a discrete-event simulation framework based on DEVS and the JavaBean component model.
- CD++ [Rod99, Wai02a] is an M&S toolkit developed in C++ that implements the original DEVS formalism.

None of the toolkits listed above, however, is capable of applying RT-DEVS to real-time embedded systems design using hardware-in-the-loop. The aim of this work is to create such a toolkit. To do so, our strategy is to reuse some of the existing CD++ software components and build new functionalities as necessary. The resultant toolkit is the

Embedded CD++ (E-CD++). In general, CD++ and E-CD++ have the following major differences:

- CD++ implements DEVS, whereas E-CD++ implements RT-DEVS (i.e., P-DEVS combined with the Time Interval Function).
- CD++ runs on workstations, whereas E-CD++ runs on a single board computer (SBC).
- CD++ uses logical time, whereas E-CD++ supports both logical and physical clock.
- E-CD++ has better simulation performance than CD++, since E-CD++ implemented a Flattened Coordinator technique to improve the performance.
- CD++ does not interact with real-world events, whereas E-CD++ does. It uses the input ports on the SBC to receive events from real input devices, such as sensors and timers. As well, the outputs can be sent through output ports connected to devices, such as motors and gears.
- CD++ can only simulate homogenous DEVS models, where as E-CD++ can seamlessly integrate DEVS components with hardware components.
- E-CD++ supports graphical modelling, which is a new functionality that CD++ does not have.

2.7 M&S Methods for RTES Design

In the past, M&S have been used to model and simulate the system under study, so that the system behaviour can be analysed and examined. M&S was used only to study the target system in a simulated environment. This work, however, attempts to apply the M&S methodology directly to the design of the target system.

Using M&S to design real-time embedded systems has a promising potential. [Ern98] stated the importance of Modelling in modern embedded system design, and [SLS00] further claimed that formal models are essential to embedded system design. However, defining a formal modelling methodology that is adequate in modelling all kinds of embedded systems is a challenge. We found that some models, such as FSM, support

event-driven reactive systems, while others target dataflow systems. A combination using both domains (e.g., telecom devices) implies simulation overhead. As well, some simulators, such as MATLAB, are adequate in modelling CVDS, but not suitable for DEDS. Due to their digital nature, embedded systems can be categorized as DEDS. Consequently, we found that RT-DEVS is an adequate state-of-the-art modelling methodology for embedded systems. Furthermore, recent advancement of the DEVS research has extended DEVS to a new embedded system design paradigm in which RT-DEVS is used as a framework that seamlessly integrates simulation models with hardware components and that also enables incremental transition from the simulated models to the actual hardware counterparts.

However, although RT-DEVS is adequate in *modelling* embedded systems, no simulation toolkit available can apply RT-DEVS directly to the *implementation* of the target system. Modelling and implementation differ in the way that, in modelling, all models (including hardware models) are simulated software components, while implementation must integrate hardware and software components. In other words, implementation must face the hardware/software partition problem, implying that no instantaneous transition exists from the modelling phase to the implementation phase.

The aim of E-CD++ is to address this issue. Since the hardware/software partition problem is NP-complete, no formal methodology can be found to solve this problem. Therefore, imperial approach has to be used instead. The E-CD++ toolkit merges the RT-DEVS formalism with hardware-in-the-loop design methodology. That is, E-CD++ creates an real-time execution environment which integrates RT-DEVS models with real hardware components. In this way, RT-DEVS is directly applied to the implementation of the system under design.

Chapter 3 Embedded CD++ (E-CD++)

In this chapter, we introduce the Embedded CD++ (E-CD++) toolkit. We first give a brief introduction to CD++ and E-CD++. We then extend our discussion to exploring four major functionalities of E-CD++: a GGAD (Generic Graphic Advanced environment for DEVS modelling and simulation) interpreter, P-DEVS simulation, the Flattened Coordinator technique, and the Time Interval Function. With the realization both P-DEVS and the Time Interval Function, E-CD++ carries out RT-DEVS simulation. Finally, performance is a critical attribute for an embedded simulator that performs real-time simulations. E-CD++ adopts the Flattened Coordinator technique to improve its performance. We provide a theoretical discussion on E-CD++ performance when exploring the Flattened Coordinator technique.

3.1 CD++

Not only does DEVS propose a framework for model construction, it also defines an abstract simulation mechanism that is independent of the model itself. This mechanism provides a high level design on DEVS framework, and it can be feasibly implemented by computer software. **CD++** [Wai02] is a simulation software which implements the DEVS simulation formalism. In CD++, two kinds of **simulators** are implemented, one for atomic and the other for coupled models. The latter is known as a **coordinator**. These simulators progress through the simulation by exchanging messages as described by the abstract simulation mechanism.

CD++ is written in C++. Two basic abstract classes exist: `Model` and `Processor`. The former is used to represent the behaviour of the atomic and coupled models, while the latter implements the simulation mechanisms. Figure 3 shows the CD++ class hierarchy.

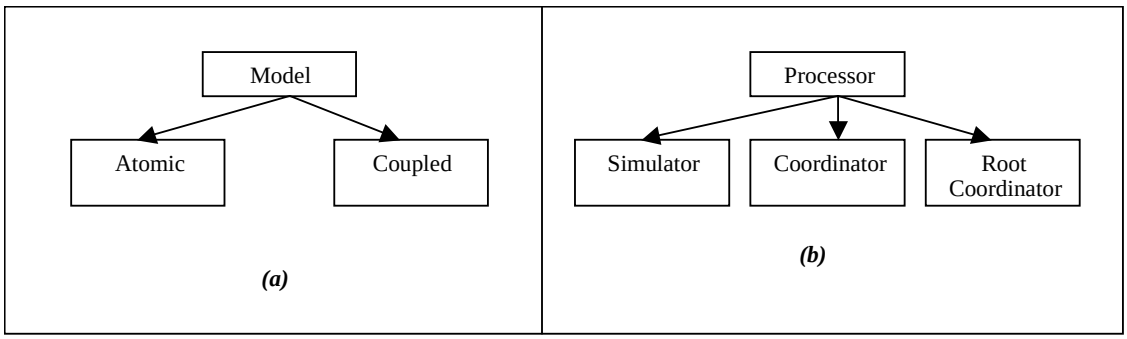

Figure 3 CD++ (a) Model hierarchy, (b) Processor hierarchy

The user-defined DEVS models are subclasses derived from the `Model` class, which defines the model states and the internal and external state transition functions. CD++ also allows the user to create a model file which defines the model hierarchy and linkage between ports.

Message passing are among `Processor` objects, which upon receiving certain messages, trigger the appropriate state transition functions.

3.2 E-CD++

CD++ was developed to run in a simulated environment carrying out only simulated results. E-CD++, by contrast, is developed to apply the RT-DEVS formalism to embedded systems design which requires real-time expansion and interact on the surrounding environment. The inputs of E-CD++ can be received by ports connected to real input devices such as sensors, timers, thermometers, or data collected from human interaction. The outputs can be sent through output ports connected to devices such as motors, transducers, gears, valves, or any other component.

E-CD++ runs on a Single Board Computer (SBC). It supports hardware-in-the-loop simulations by developing hybrid hardware/software systems -- integrations of simulated software models and real hardware components [LPW03].

The DEVS model is then loaded onto the SBC running E-CD++ for validation. E-CD++ supports RT-DEVS in which time advancement is driven by the wall-clock. Furthermore, the inputs and outputs ports on the SBC are real hardware that is used by the E-CD++ to let the DEVS model interact with the environment. E-CD++ can also integrate simulation models with hardware components, which enables incremental transition from the simulated models to the actual hardware counterparts.

The testing results obtained on the SBC can be compared with the simulation results obtained on the host workstation. If the two results do not agree, then the DEVS model or the event file developed on the workstation can be easily modified or adjusted to obtain more accurate results.

E-CD++ inherits all the functionalities of the original CD++, while adding the following new functionalities:

- Supports the GGAD graphical modeller;
- Implements RT-DEVS by implementing the P-DEVS formalism and the time interval function;
- Implements the Flattened Coordinator technique;
- Is able to simulate DEVS models in an embedded computing environment with limited resources;
- Is able to let DEVS models respond to real world events via input and output ports of the embedded system.

3.3 GGAD Graphical Notation

E-CD++ provides a graphical user interface (GUI) for modellers to specify atomic models graphically, enabling non-expert users to define atomic models in a easier and more intuitive way. The tool generates textual specifications of the models represented graphically in the GUI.

The GUI is based on a DEVS-graph notation presented in [HSKP97], which allows defining atomic models using a graphical modelling tool. An atomic model is placed

inside a box (Figure 4). An external state transition is represented by a dotted line in which the input event is represented by "?". Similarly, an internal state transition is represented by a solid line in which the output event is represented by "!". For example, an input event *in?m* means that a message *m* arrives at the input port *in*, and an output event *out!m* means that a message *m* is run through port *out*. The input and output ports are denoted by the black triangles. The atomic states are marked by circles in which the state names and the time advance functions are defined.

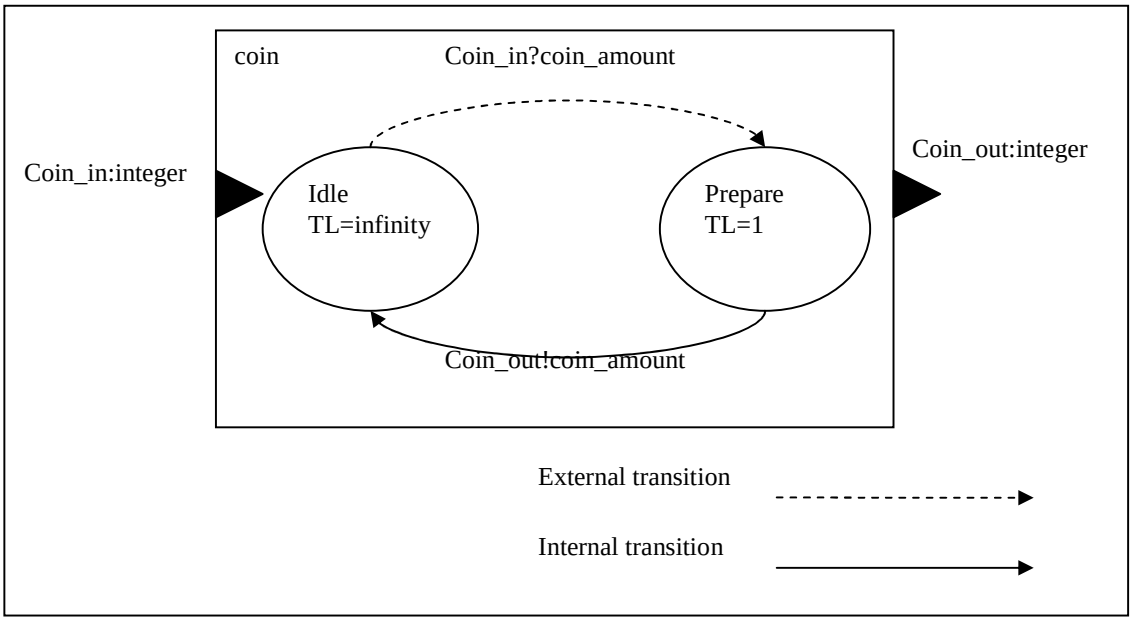

Figure 4 Graphical definition of an atomic model: Coin Displayer

Figure 4 is a graphical representation of an atomic model called *coin*. This model simulates the behaviour of the coins displayer in a vending machine. It has one input port, namely *coin_in*, representing the input coins slot and one output port called *coin_out* which represents the display of the inserted coins' amount, and the port values of both ports have integer data type. When coins are inserted into the coins slot, it takes one second for the coins displayer to display the coins' amount.

The two circles in the diagram represent the model's two internal states whose names and time intervals are defined inside the circles. The "Idle" state is the initial state of which the time interval is set to infinity, which implies that the model may remain idle if no external events (e.g., inserting coins) arrives. In contrast, the "Prepare" state has only one

second time interval, meaning that it takes one second for the “Prepare” state to change to another one. That is, when coins are inserted into the coins slot, it takes one second for the coins displayer to display the coins’ amount.

The external transition function of the coins displayer is defined by the dotted arrow line in Figure 4. When a coin is inserted via the input port, the external transition function changes the model’s state from “Idle” to “Prepare” and stores the input value to the local variable “coin_amount”. After one second time interval elapses, the output function, defined right below the solid arrow line, outputs the “coin_amount” value to the output port, and the internal state transition, defined by the solid line, changes model state back to “Idle”.

The textual specifications are defined by a modelling language called GGAD. A GGAD file is a text file that contains an atomic model written in GGAD. As an example, Figure 5 provides the textual definition of the atomic model represented in Figure 4.

```
[coin]
in: coin_in
out: coin_out
var : coin_amount
state: idle prepare
initial: idle
int: prepare idle coin_out!coin_amount
ext: idle prepare any(coin_in)?1 {coin_amount = coin_in;}
idle: infinite
prepare: 0:0:1:0
coin_amount: 0
```

Figure 5 GGAD textual definition of the coin model

This first line in the GGAD file is always the name of the atomic model encoded by the square brackets. The rest of the file content defines the model’s ports, states, state transition functions, local variables used in transition functions, and time advance functions. The order of these definitions is arbitrary, and tokens in GGAD are separated by white spaces.

- The input ports (line 2) are defined by the keyword “in” followed by a colon and a list of port names separated by white spaces.

- Similarly, the output ports (line 3) are defined by the keyword “out” followed by a colon and a list of output port names.
- The local variable list (line 4) and the state list (line 5) are both followed the same syntax rule, except their responding keywords are “var” and “state” respectively.
- The keyword “initial” is used to define the initial state. For example, the initial state of the coin model (line 6) is “idle”.
- The time advance function is defined by a state name followed by a colon and a GGAD time. The GGAD time has the format “HH:MM:SS:MS” For example, the elapse time of the prepare state is 1 section (line 10). In addition, the keyword “infinite” defines the passive state. In our example, the “idle” state is a passive state (line 9).
- The internal transition function is defined as follows. It starts with the keyword “int” followed by a colon. The next two tokens are the start state and the destination state. Then remainder part of the function defines the output function. The keyword “!” is the output mark. The left hand side of “!” is the output port name, and the right hand side is the output value. For example, the internal transition of coin (line 7) starts at state “prepare” and ends at state “idle”. The output function outputs the value of local variable “coin_amount” to port “coin_out”.
- The keyword “ext” denotes the start of an external transition function (line 8). The external transition function also has a start state and a destination state, which are defined in the same syntax as that defined in internal transition functions. The keyword “?” is the so-called input mark. The left hand side of “?” is a GGAD expression, while right hand side is an integer constant. Once an external event takes place, the GGAD modeller evaluates the expression and compares the result with the integer constant. If the two values agree, the GGAD modeller will execute this external transition function. Otherwise, the function will not be executed. GGAD uses this approach to select the correct external transition function to execute upon a particular external event. In our coin example, the expression “any(coin_in)” returns 1 if a new value arrives at input port “coin_in” and returns 0 otherwise. So, the external transition defined is line 8 will be

executed upon the event where new coins are inserted into “coin_in”. Detailed exploration on GGAD expressions will be covered in the next chapter.

The last part of the external transition function is a list of GGAD actions, enclosed by curry brackets. They form the body of the external transition function. GGAD actions are C++ assignment like statements and are separated by semicolons. In the coin model example, the GGAD action “coin_amount = coin_in;” assigns the input value arrived at input port “coin_in” to the local variable “coin_amount”. GGAD actions will be discussed in the next chapter in detail.

The complete definition to GGAD grammar can be found in Appendix B: Grammar for GGAD Models.

3.4 P-DEVS Simulation Algorithms

The formal definition and semantics of P-DEVS is given in Chapter 2. This section discusses the definition of P-DEVS simulation algorithms used in E-CD++. That is, we focus on how to transform the specifications of P-DEVS (written in mathematical terms) into algorithms that can be implemented by computer programs.

The P-DEVS formalism allows the modeller to specify the state transition behaviours of atomic and coupled models, as well as the port connections among models. These connections constitute the model hierarchy. The distinctiveness of P-DEVS is that it supports parallel executions on simultaneous state transitions of atomic models and concurrent handling of simultaneously scheduled external events. Moreover, handling external events from the environment, executing transition functions, and exchanging messages among models through their input and output ports that trigger more state transitions to happen constitute all the activities of P-DEVS simulations. More specifically, the following three functionalities are needed:

1. *Parallel executions on simultaneous state transitions*. This functionality may be achieved by periodically determining the imminent atomic models at each simulation cycle with time advancement and synchronizing their state transition activities. In E-CD++, each model has internal variables to keep the time of the simulation, as listed below.

   t_L *Time of last transition*

   t_N *Time of next transition*

   An **imminent atomic model's simulator** has the smallest value of t_N. among its siblings. In other words, it is a model that holds an imminent state transition that will occur in the next simulation cycle in which time will advance to the model's t_N. A simulation cycle advances time from t_L to t_N, where t_L is the end time of the previous simulation cycle and t_N is the finish time of the current cycle. Within this time period, there may be multiple imminent atomic models. With P-DEVS implementation, E-CD++ is capable of executing these state transitions simultaneously. This capability differentiates E-CD++ from the original CD++ simulator, where the SELECT functions are used to serialize the executions of these simultaneous state transitions.

   To achieve parallelism on state transitions, synchronization of models' activities is necessary. The synchronization of atomic models' state transitions can be achieved by inter-component messaging. E-CD++ implements a new inter-component messaging architecture that is very different from the original CD++ design, so that it can offer this functionality. Two main categories of messages exist: **synchronization** and **content** messages. Each of these categories consists of several types of messages.

   Synchronization messages:

   @ *Collect message*

   ******* *Internal message*

   done *Done message*

Content messages:

Q *External message*

Y *Output message*

The Content Messages are used to exchange data among components via their input and output port connections. The concept of Synchronization Messages is newly introduced by E-CD++. Each coupled model in E-CD++ maintains an important data structure called **Synchronization Set**, which is a subset of its children that are scheduled to have state transitions in the next simulation cycle. That is, they are imminent components. Synchronization Messages are exchanged among components in order to periodically create, update and clear synchronization sets in each simulation cycle.

2. *Handling simultaneous external events in parallel*. In the original DEVS formalism, the atomic model can handle only one external event at a time. E-CD++ overcomes this limitation by redesigning the external transition function. Rather than invoking δ_{ext} immediately upon an arrival of an external event, E-CD++ stores the external message in the receiving model's **Message Bag**, which is a set of external messages and then adds this model to the Synchronization Set. The model's external transition function, once invoked, processes all messages in the Message Bag altogether, as opposed to one single external message at a time. In this design, simultaneous external events that take place within a simulation cycle, i.e., between $\boldsymbol{t_L}$ to $\boldsymbol{t_N}$, are handled by δ_{ext} in parallel.

3. *Ability to resolve conflicts caused by simultaneously scheduled internal and external state transition within one atomic model*. This functionality is achieved by implementing the confluent function δ_{con}, which is a new device introduced by P-DEVS.

In E-CD++ implementation, the **Simulator** is responsible of invoking the atomic model's $\lambda(s)$, δ_{ext}, δ_{int}, δ_{con} functions, while the **Coordinator** is responsible for the simulation of

coupled models. Both simulators and coordinators are capable of sending and receiving messages. Their implementations are described below. The algorithms that follow are based on that in [Cho94b], with minor modifications.

According to Cho's algorithm, when a simulator receives a (@, *t*) message (Figure 6), it executes the atomic model's output function λ (line 3) and sends the output to the parent coordinator (line 4). Note that the simulator executes λ at exactly t_N time (line 2), ensuring the correctness of the simulation. Finally, it sends out the Done message to its parent, indicating the completion of the execution (line 5).

```
SIMULATOR
1.  when a ( @ , t ) message is received
2.      if t = t_N then
3.          y := λ(s)
4.          send ( y , t ) to the parent coordinator
5.          send ( done, t ) to the parent coordinator
6.      end if
7.      else raise error
8.  end when
```

Figure 6 Simulator Receiving Collect Message

When a simulator receives an External Message (Figure 7), it simply adds it to the Message Bag (line 3) and does so atomically (line 2 and 4) to avoid race conditions in the concurrent computing environment.

```
SIMULATOR
1.  when a ( q , t ) message is received
2.      lock the bag
3.      Add event q to the bag
4.      unlock the bag
5.  end when
```

Figure 7 Simulator Receiving External Message

However, there is a minor difference between Cho's algorithm and ours in handling the External Message. Cho's algorithm also sends a (done, t) message to the simulator's

parent coordinator after the event is added. We found that generating the done message is not necessary, provided that the message handlers for the synchronization messages (namely @ and *) send done messages. That is, there is no need for content message handlers to send done messages. We have proved this point by statement.

Figure 8 outlines Cho's algorithm of the Simulator's Internal Message handler. The arrival of the *(*, t)* message indicates that an atomic model's transition function must be executed. The transition function to be executed will depend on the current time, *t*, and the content's of the Message Bag. If $t < t_N$, then it is not the time for an internal transition, and it must be the case that the Message Bag is not empty, and δ_{ext} is executed, consuming all the external messages in the Message Bag at once.

```
SIMULATOR
1.  when a ( * , t ) message is received
2.          case t_L ≤ t < t_N
3.                  e := t - t_L
4.                  s := δ_ext( s, e, bag )
5.                  empty bag
6.          end case
7.          case t = t_N and bag is empty
8.                  s := δ_int( s )
9.          end case
10.         case t = t_N and bag not is empty
11.                 s := δ_con( s, bag )
12.                 empty bag
13.         end case
14.         case t > t_N or t < t_L
15.                 raise error
16.         end case
17.         t_L := t
18.         t_N := ta (s)
19.         send ( done, t_N) to parent coordinator
20. end when
```

Figure 8 Simulator Receiving Internal Message

If $t = t_N$,, it is the time for an internal transition. If no external messages has been received, then δ_{int}, is executed, but if there are external messages in the Message Bag, then δ_{con} is called instead. All other cases are considered as errors.

Once the appropriate transition function is executed, the simulator update its t_L to current time *t*, and t_N to *ta(s).*

We now describe the behaviour of *Coordinator*. A coordinator is responsible for the simulation of a coupled model. It executes the translation function that translates output events to input events, maintains the Synchronization Set which stores the imminent children, and synchronize its children's state transition by sending out Synchronization Messages.

Figure 9 illustrate how a Coordinator handles the Collect Message. (This is also Cho's algorithm) Fist of all, it checks if the Collect Message is received exactly at time t_N (line 2). If not, it raises an error (line 11). It then updates its t_L to *t* (line 3) and sends (*@*, *t*) to all of its imminent children (line 5). The imminent children can now be stored in the Synchronization Set (line 6), which implies that their state transitions are scheduled to take place in the next simulation cycle when time advances to t_N. After it receive the Done Message from all it imminent children (line 8), the Coordinator sends a Done Message to its parents indicating the completion of the task (line 9).

```
COORDINATOR
1.  when a ( @ , t ) message is received from parent coordinator
2.      if t = t_N then
3.          t_L := t
4.          for all imminent child processors i with minimum t_N
5.              send ( @, t ) to child i
6.              cache i in the synchronize set
7.          end for
8.          wait until ( done, t )'s have been received from all imminent processors
9.          send ( done, t ) to parent coordinator
10.     end if
11.     else raise error
12. end when
```

Figure 9 Coordinator Receiving Collect Message

When a Coordinator receives an Output Message (Figure 10), the message must be sent from one of its children, because in E-CD++ implementation, Output Messages are only sent upwards in the models hierarchy from children to their parents.

```
COORDINATOR
1.  when a ( y , t ) message is received from child i
2.      for all influencees, j of child i
3.          q := z_{i,j} ( y )
4.          send ( q, t ) to child j
5.          cache j in the synchronize set
6.      end for
7.      if self ∈ I_i ( y is to be transmitted upward) then
8.          y := z_{i, self} ( y )
9.          send ( y, t ) to parent coordinator
10.     end if
11. end when
```

Figure 10 Coordinator Receiving Output Message

The first action that a coordinator performs upon the arrival of an Output Message is to invoke the translation function to translate the Output Message into the External Message and send it to all of its receiving models (line 2 – 4). Then the Coordinator caches the receiving models into the Synchronization Set (line 5), implying that their external state transitions are scheduled to occur in the next simulation cycle.

If, however, the coordinator itself is also one of the receiving models (line 7), it means that the Output Message needs to be forwarded upward to its parent. In this case, the coordinator generates another Output Message and sends it to its parent coordinator (line 8 & 9).

The coordinator handles External Messages in the same way as a simulator does. That is, it atomically adds the incoming message to its Message Bag (Figure 11).

```
COORDINATOR
1.  when a ( q , t ) message is received from parent coordinator
2.      lock the bag
3.      Add event q to the bag
4.      unlock the bag
5.  end when
```

Figure 11 Coordinator Receiving External Message

The behaviour of a coordinator receiving an Internal Message is illustrated in Figure 12. When a coordinator receives an Internal Message, it first checks whether the current time, *t*, is somewhere in between the last transition time and the next scheduled transition (line 2). If not, it returns an error (line 20). If the time is right, it processes all the External Messages stored in the Message Bag by translating them into new External Messages (line 5) and sending them to the receiving components (line 6). These receiving components are cached into the Synchronization Set, for they need to process the External Messages that have just been sent to them in the next simulation cycle. Having processed all the External Messages, the coordinator empties the Message Bag (line 10). Next, the coordinator sends out an Internal Message to every component saved in the Synchronization Set (line 11 – 13), to trigger the state transitions of these imminent children. Then, the coordinator is blocked waiting for the Done Messages from all the imminent children which have just received the Internal Messages (line 14).

```
COORDINATOR
1.  when a ( * , t ) message is received from parent coordinator
2.      if t_L ≤ t ≤ t_N
3.          for all q ∈ bag
4.              for all receivers of q, j ∈ I_self
5.                  q := z_self,j (q)
6.                  send ( q, t ) to j
7.                  cache j in the synchronize set
8.              end for
9.          end for
10.         empty bag
11.         for all i in the synchronize set
12.             send ( *, t ) to i
13.         end for
14.         wait until all ( done, t_N)'s are received
15.         t_L := t
16.         t_N := minimum of components' t_N's
17.         clear the synchronize set
18.         send ( done, t_N ) to parent coordinator
19.     end if
20.     else raise an error
21. end when
```

Figure 12 Coordinator Receiving Internal Message

When all the Done Messages are arrived, the coordinator unblocks itself and updates its t_L and t_N, and clears its Synchronization Set (line 15 – 17). And finally, it sends the Done Message to its parent (line 18).

The last piece of work needs to be explained in P-DEVS is the *Root Coordinator*. It is a special processor that is above the topmost coordinator. It is responsible for driving the simulation and advancing the virtual simulation time. Figure 13 represents the algorithm of the Root Coordinator. First, it advances the simulation time to t_N of the topmost coordinator (line 1). This implies that the state transition of the topmost coordinator may occur right away. Then, the Root Coordinator keeps sending the Collect Message and the Internal Message to the topmost coordinator (line 3 – 6) as well as updating the simulation time (line 7).

ROOT COORDINATOR

```
1.  t := t_N of the topmost coordinator
2.  while t ≠ ∞ or more external events to come
3.      send ( @ , t ) to the topmost coordinator
4.      wait until ( done , t ) message is received from it
5.      send ( * , t ) to the topmost coordinator
6.      wait until ( done, t_N ) message is received from it
7.      t := t_N of topmost coordinator
8.      if external event arrives
9.          send ( q, t ) to the topmost coordinator
10.     end if
11. end while
12. raise simulation completed
```

Figure 13 Root Coordinator Behaviours

The Root Coordinator can also handle external events. These events may be stored in an events file which contains a sorted queue of events. When an external event occurs, the Root Coordinator sends an External Message to the topmost coordinator (line 8).

The simulation completes when *t* becomes infinity and there are no more external events left.

3.5 Flattened Coordinator Technique

Real-time applications require the simulation software be able to carry out results within specified time constraints. Therefore, performance is essential to the success of a real-time simulation software. The Flattened Coordinator technique [Kim00] is introduced to improve the performance of E-CD++. The original CD++ needs to build a DEVS model hierarchy. As shown in Figure 14, CD++ builds a same hierarchy for coordinators and simulators as that for DEVS models. When a DEVS model executes, one simulator object is created for each atomic component, and one coordinator object for each coupled component in the hierarchy.

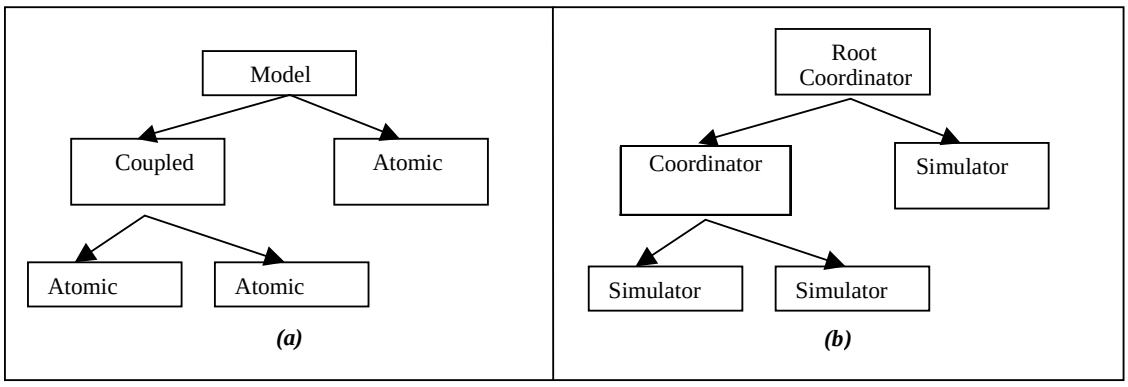

Figure 14 CD++ (a) Model hierarchy, (b) Processor hierarchy [Gli04]

The problem with the hierarchical approach is that performance is not scalable. As the size and complexity of DEVS models grow, so is the processor hierarchy, resulting in the reduction of the simulation performance. The simulation algorithms explained in Section 3.4 revealed how messages are generated and exchanged among coordinators and simulators. Those algorithms showed that the number of messages exchanging among processors is proportional to the complexity of the processor hierarchy, which is measured by the number of processors on the hierarchy. In other words, as the hierarchy grows, so is the performance overhead incurred by messaging. In order to optimize the performance, therefore, the simulator needs to reduce the complexity of the processor hierarchy, i.e., the number of processors.

This is the main concept behind the Flattened Coordinator technique. Flattened Coordinator, therefore, simplifies the simulation hierarchy by eliminating the coordinators in the hierarchy and by making direct messaging communications between the Flattened Coordinator and the simulators. Both the model and the simulation hierarchies for this case are shown in Figure 15. A similar development for other DEVS simulators can be found in [Gli02] and in [Kim00].

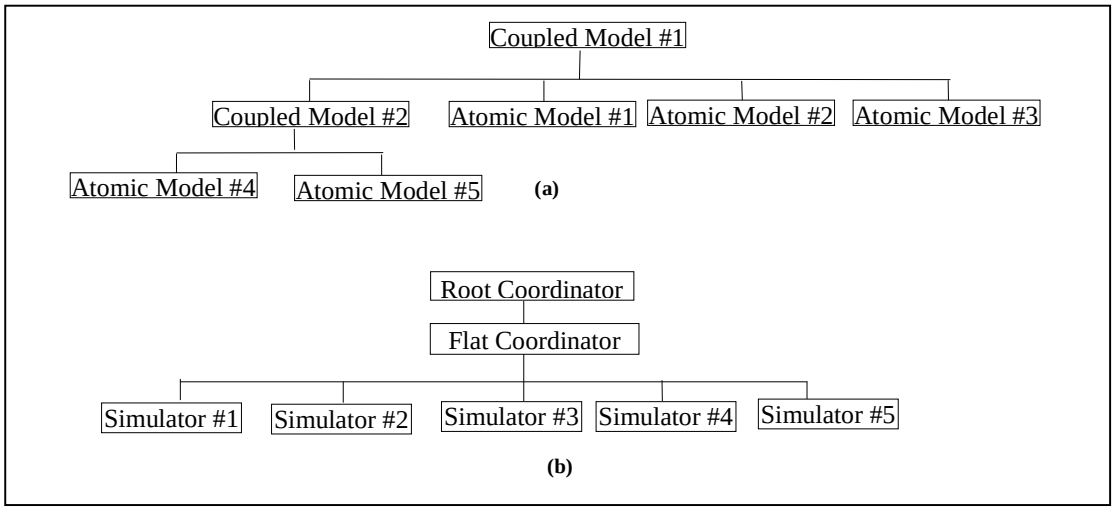

Figure 15 Flattened Coordinator Technique
(a) Example of a model hierarchy, (b) Associated processor hierarchy

The Flattened Coordinator transforms the hierarchical structure of the model to a flattened structure by eliminating coordinators. The transformation, however, must preserve the original port linkage relationship among atomic models, so that the correctness of the simulation does not suffer. In order to achieve this, the Flattened Coordinator technique needs to rewire the model port links to bypass the coordinator ports. Consider, for example the model hierarchy shown in Figure 15 (a). Suppose Atomic Model #1 needs to send a message to Atomic Model #4. The message will first be sent to Coupled Model #2, which will then forward the message to Atomic Model #4, as shown in Figure 16 (a). When Coupled Model #2 is eliminated, however, the Flattened Coordinator must rewire the port links of Atomic Model #1 and Atomic Model #2, so that messages can still be exchanged between them (Figure 16 (b)).

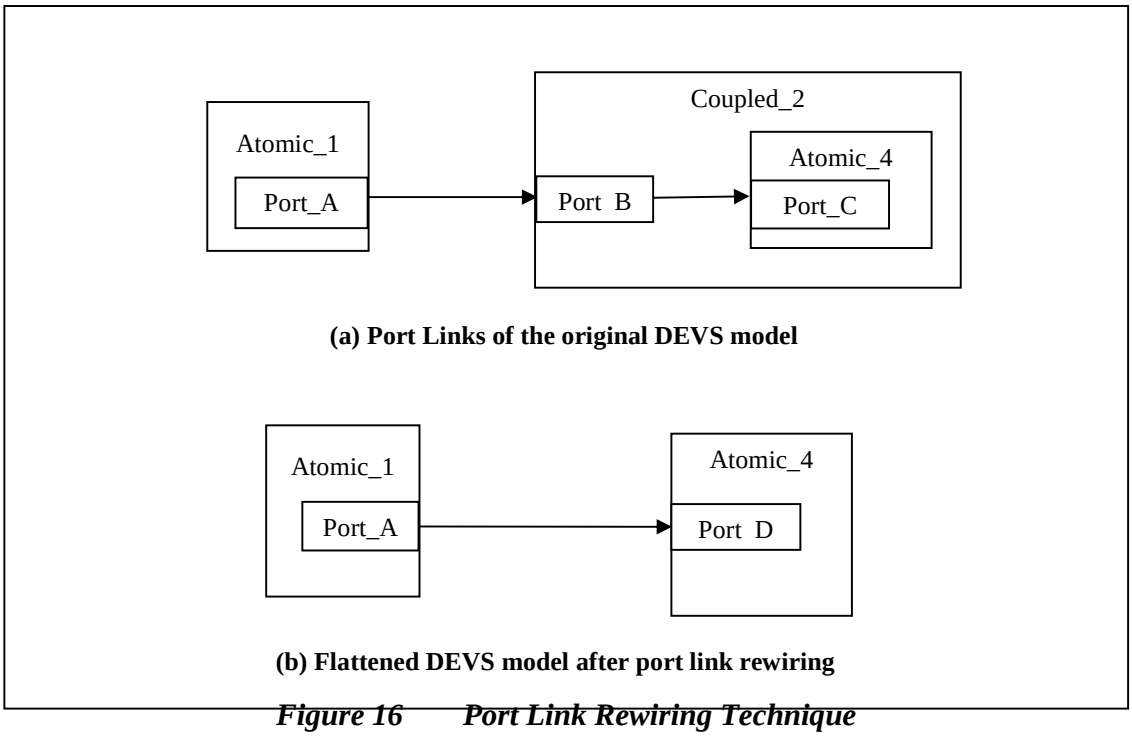

Figure 16 Port Link Rewiring Technique

Furthermore, as a result of the elimination of coordinators, the Flattened Coordinator must receive and send messages directly from and to the Root Coordinator in order to carry out the simulation process.

Since the performance of E-CD++ is directly related to the efficiency of its messaging, the performance can be quantitatively rated by the number of messages generated during the simulation cycle, which is proportional to the number of Processors. Therefore, the performance gain of the Flattened Coordinator technique can be measured by the reduction rate of the Processors. That is, the performance improvement ratio of the Flattened Coordinator technique (R), is one minus the number of processors on the flattened structure (P_f) divided by that on the original non-flattened hierarchy (P_o).

$$R = 1 - \left(\frac{P_f}{P_o} \right)$$

This formula is also verified by real simulation experiments in Chapter 5.

To illustrate this calculation numerically, we consider that a non-flattened processor hierarchy is organized as an *n*-ary tree with *h* levels. Then, the total number of Processors on this tree (P) is given by the following formula:

$$P = \sum_{i=0}^{h} n^i = \frac{n^{h+1} - 1}{n - 1}$$

The number of processors on the flattened hierarchy, in contrast, is equal to the number of leaves (i.e., simulators) plus the Flattened Coordinator, which is $n^h + 1$. Therefore, the performance improvement ratio (R') of flattening a processor hierarchy that is a full *n*-ary tree with *h* levels is:

$$R' = 1 - \frac{(n^h + 1)(n - 1)}{n^{h+1} - 1}$$

For instance, performance after flattening a full ternary tree (an n-ary tree with n = 3) with 2 levels is improved by 1 - 10 / 13 = 23%.

3.6 Time Interval Function

One major difference between evaluating the correctness of solutions developed in the simulation world and that in the real world is that the former are evaluated in the virtual time, whereas the latter often bind to real-time constraints. A *real-time system* is defined as a one whose correctness depends not only on the computational results, but also on the time at which the results are produced [Sta88, Sta96]. If a system delivers the correct answer after a certain deadline, it could be regarded as an unsuccessful response. Consequently, a real-time simulator must offer the functionality of the *Time Interval Function* where time constraints can be stated and validated. E-CD++ offers this functionality by implementing the *wall-clock time advancement* and the *event deadline checking*.

In order to run real-time simulations, advance of the simulation-clock is tied to the wall-clock (*i.e.* physical time). The *Root Coordinator* object provides this functionality. The

Root Coordinator manages the time advancement along the execution of a simulation. It is also responsible for starting very new simulation cycle. When the *virtual time* approach is used, after a simulation cycle finishes, the logical clock time is incremented to the next scheduled event by the Root Coordinator without any physical delay. That is, the Initialization Messages are immediately generated and sent by the Root Coordinator to start a new simulation cycle. For the real-time simulation, however, the Root Coordinator must wait until the physical time reaches the next event time to initiate the new cycle. This implies that the periods of inactivity must not be skipped. The simulation process remains quiescent while these periods are being experienced. Instead of logically advancing the virtual time up to the next scheduled event (as what's done by the virtual time approach), the Root Coordinator expects the scheduled wall-clock time to be reached and only then starts the new simulation cycle. In other words, a new simulation cycle can be started either due to the reception of an *external event*, or due to the consumption of the time indicated by *ta(s)* of the Root Coordinator. Hence, messages interchanged between processors are sent at their actual scheduled wall-clock time.

E-CD++ creates a state machine to implement wall-clock time. The implementation uses standard UNIX timer facilities provided by the `<linux/time.h>` library. Figure 17 illustrates a state machine for this timing behaviour.

- The state machine's starting state is the "Inactive state", in which E-CD++ is passive. When E-CD++ reads in the external events file or when new events arrive (E1), the Root Coordinator calls `add_timer()`to create timers with the associated expiry timestamps for all the external events (A1a), and then it calls `interruptible_sleep_on()` (A1b) to transfer the state to the "Timer Counting-down state", in which E-CD++ remains passive until timer expiry.

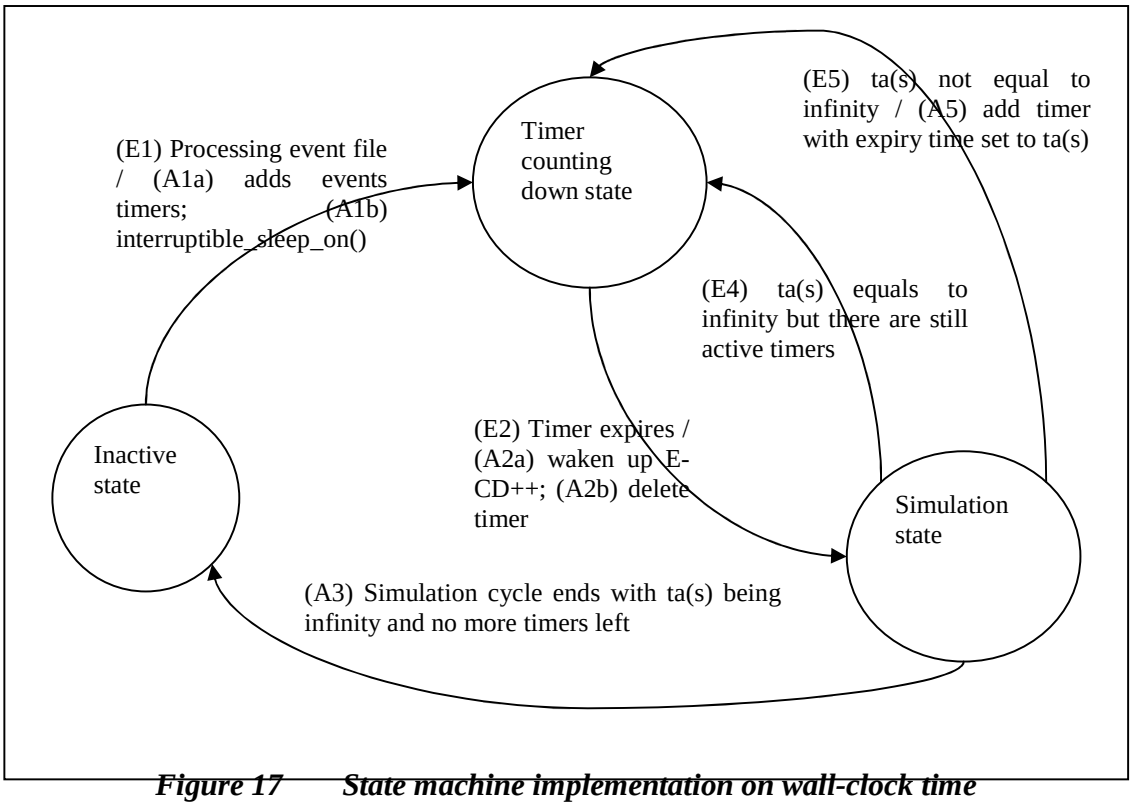

Figure 17 State machine implementation on wall-clock time

- As the wall-clock time advances, those timers will expire at the moments when the external events arrive (E2), which will invoke their timeout functions. The timeout functions will call `wake_up_interruptible()` to wake up E-CD++ (A2a) to transfer to the "Simulation state", and will also call `del_timer_sync()` to delete the timer associated with the arrived external event.

- In the Simulation state, a new simulation cycle is started, and a new ta(s) is calculated. If the new ta(s) is set to infinity, and if there are no more timers left, the Root Coordinator will call `interruptible_sleep_on()` to go back to the "Inactive state" (A3) (i.e., deactivating E-CD++). E-CD++ will be waken up upon the arrival of the next external event.

- If ta(s) is infinite, but there are still active timers (E4) (e.g., there are still some scheduled external events), then Root Coordinator will move the state from "Simulation" to "Timer Counting-down".

- If, however, the new ta(s) calculated in the Simulation state does not equal infinity (E5), then the Root Coordinator will create a new timer with the expiry timestamp set to *ta(s)*, and will move the state to "Timer Counting-down state" (A5), so that E-CD++ will be waken up again after *ta(s)* time.

Timeliness along a simulation is a substantial property in the real time approach. In a typical real-time situation, the model has to react to an external event and generate the output within a given time in order to solve a given problem. When a model is executed in real-time simulation, it is important to check different time constraints along the simulation. Particularly, the time at which an event has been completely processed is a meaningful measure of success.

During the simulation cycle, E-CD++ validates the time constraints which are stated by the Time Interval Function. The Time Interval Function specifies the deadline before which a simulation cycle must complete (e.g., An output must be arrived at a certain output port.). E-CD++ allows the modeller to indicate the deadlines for external events. E-CD++ checks the wall-clock time at which the simulation of the external event is finished against the specified deadline. If the completion time is later than the deadline, it indicates the failure.

E-CD++ creates a new format of the event file in which the deadlines are specified. The new extended format of the **event file** is illustrated in Figure 18.

Event time	***Associated deadline***	***input port***	***associated output port***	***value***
hh:mm:ss:mseg	hh:mm:ss:mseg	port name	port name	numeric value

Figure 18 Format of the event file in the real time extension

Figure 18 shows that not only an *associated deadline* but also an *output port* must be indicated in the new event file format. As a result, the simulator can check whether the physical time meets the associated deadline when sending an output through the associated port. Once the execution has finished, both successful and unsuccessful deadlines are stored for further study of the simulation process.

When loading the event file, the E-CD++ simulator stores deadlines and their associated output ports into a list of <deadline, port> pairs, named *deadlineList*. When the Root Coordinator receives an Output Message, it searches through the deadlineList to fetch out the <deadline, port> pair of which the port number matches that in the Output Message. It then compares the wall-clock time with the deadline. Figure 19 provides the pseudo code of this algorithm.

```
ROOT COORDINATOR
parse the event file and create Deadlines List (list of <deadline, port> pairs)
when a ( y , t ) message is received from TOP coordinator
        for each <deadline, port> in Deadline List
                if  port == outport in y then
                        if deadline >= wall-clock-time then
                                output value to port and mark simulation as successful
                        else
                                output value to port and mark simulation as unsuccessful
                        end if-else
                        delete <deadline, port> from Deadline List
                        quit loop
                end if
        end for loop
end when
```

Figure 19 Deadline checking algorithm

Chapter 4 E-CD++ Software Architecture

In the previous chapter, we explained *what* E-CD++ could do. In this chapter, we will discuss *how* these functionalities are designed and implemented. First of all, the Main Simulator manages the general aspects of the simulation. The Modelling subsystem constructs the DEVS model hierarchy. An important component of the Modelling subsystem is the GGAD Model Loader that supports the graph-based notation, introduced in section 3.3. The Simulation subsystem implements Cho's algorithms [Cho94b] for simulators and coordinators, which were outlined in the previous chapter. The subsystem also includes special coordinators including the Flattened Coordinator and the Root Coordinator. Furthermore, The Modelling subsystem and the Simulation subsystem are the major components that carry out the P-DEVS implementation. The last subsystem we will discuss is the Messaging Subsystem, which is responsible for delivering various types of messages.

4.1 E-CD++ software architecture overview

The E-CD++ software architecture is object oriented. The software is modularized as a group of components that have well-defined behaviours and have relatively independent functionalities. E-CD++ software consists of the following major components:

- Main Simulator
- DEVS Modelling Subsystem
- Simulation Subsystem
- Messaging Subsystem

Figure 20 illustrates the interactions among these software components.

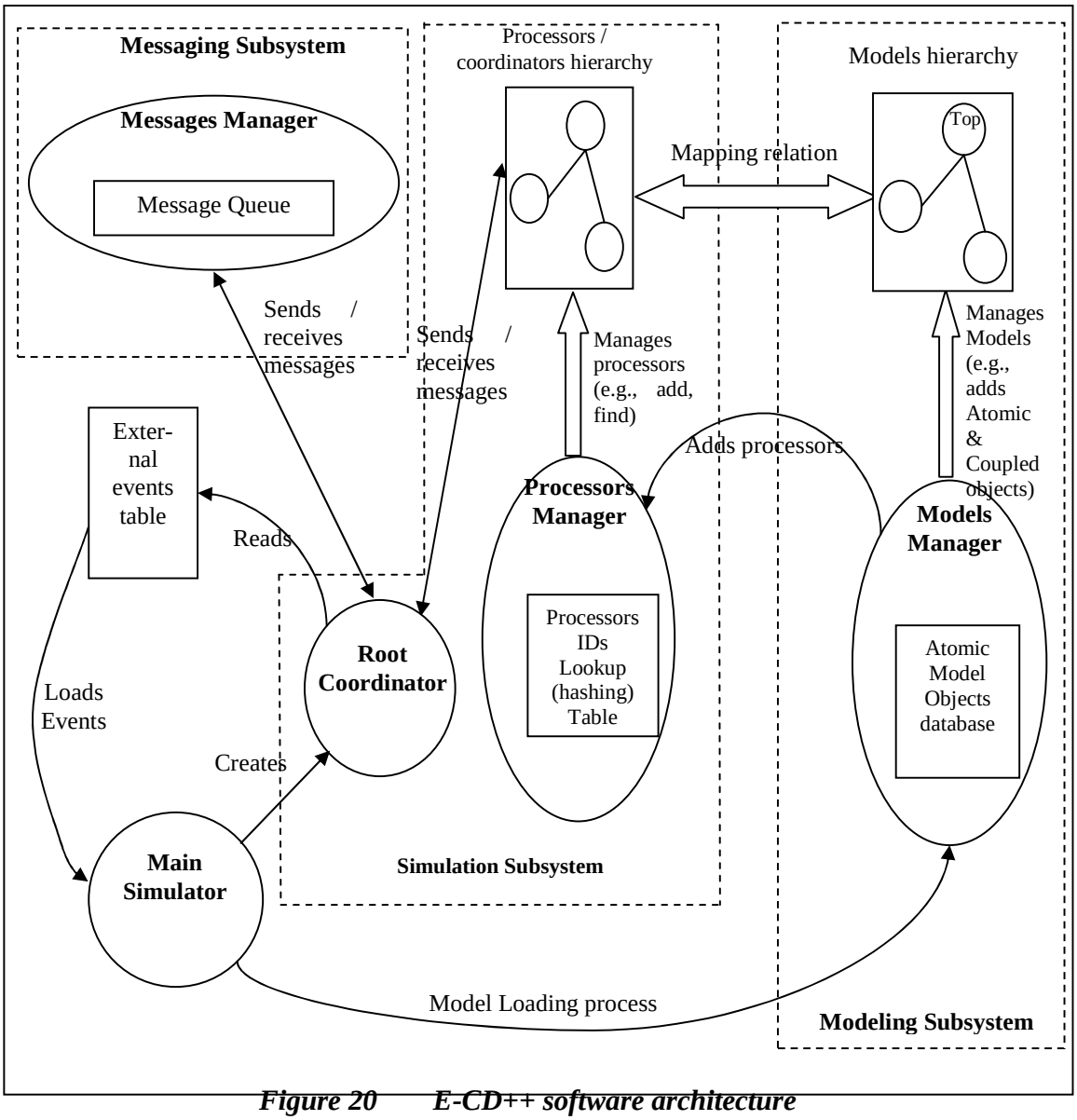

Figure 20 E-CD++ software architecture

The high-level design walk-through is summarized in the rest of this section. This walk-through provides an overview of the simulation event sequence and also explains the interactions among the subsystems. The detailed design of each subsystem is discussed in the following sections.

1. The Main Simulator (MS) is the very first object created when E-CD++ starts. The constructor of the MS, being called when the MS object is instantiated, performs the Atomic Models Registration, which adds function pointers that point

to the constructors of all the user-defined atomic models' classes into a Models Manager's table (a hashing table that serves as the Atomic Models Objects Database). These atomic models will be instantiated during the Models Loading process, which is the next step performed by the MS.

2. After the Atomic Models Registration is performed, the MS constructs the DEVS models hierarchy. The MS parses the user-defined DEVS models file in which the DEVS components and their inter-relations are defined (e.g., atomic and coupled models, ports, links, states, etc.).

3. While the MS parses the models file, it calls the Models Manager and the Processors Manager to construct two tree-like data structures in parallel. The first is the Models Hierarchy Tree, and the second is the Simulators/Coordinators Hierarchy Tree.

   The nodes of the Models Hierarchy Tree belong to the Model class, which has two subclasses: `Atomic` and `Coupled`, representing atomic and coupled models respectively. Every node on the Models Hierarchy Tree is instantiated either as an Atomic object or as a `Coupled` object. The non-leafs nodes of the Models Hierarchy Tree represent Coupled models, while the leaf nodes are Atomic models, which are subclasses derived from the Atomic class and whose behaviours are defined by the user-defined classes. The Coupled model object contains a data member called "children" which is a list of its children's models IDs. As well, each Coupled or Atomic object contains a list Ports objects which specify the input and output relationships among the models. The top node of the Models Hierarchy Tree is a special Coupled node called "Top". After all the nodes are loaded, the resulting Models Hierarchy Tree represents the model hierarchy defined by the input model file.

   If the Flattened Coordinator technique is enabled, two extra actions are taken during the Models Hierarchy Tree construction: (1) all the Coupled model objects

are eliminated from the tree, and (2) all the Atomic model objects' port links are rewired to bypass Coupled models.

The Models Hierarchy Tree and the Simulators/Coordinators Hierarchy Tree are constructed in parallel. That is, when the Models Manager adds a Coupled or an Atomic node to the Models Hierarchy Tree, the Processors Manager adds a Coordinator or a Simulator to the Simulators/Coordinators, correspondingly. The nodes on the two trees, therefore, have one-to-one mapping relationship. Note that if the Flattened Coordinator technique is used, no Coordinator objects (except the Top Coordinator) is created, since all the Coupled models are eliminated from the Models Hierarchy Tree.

4. After the Models Hierarchy Tree and the Simulators/Coordinators Hierarchy Tree are constructed, the MS loads the External Events File, if there is one, and creates the Root Coordinator and calls its simulate() function. The Root Coordinator manages the global aspects of the simulation. It receives the external events either from the pre-defined External Events File or from the real world via the physical input ports on the embedded computer. It also manages the time advancement for the simulation cycle. If a stop time is defined, the Root Coordinator terminates the simulation cycle when the time is reached.

5. The Root Coordinator also generates the very first message in the simulation, which triggers other processors and coordinators to receive and send messages. The Message Manager is responsible for messages delivery throughout the simulation cycle. It manages a Message Queue, where messages are received and sent in a FIFO (First-In-First-Out) order.

6. The simulation cycle continues by simulators and coordinators keeping sending and receiving messages among each other, and atomic models' transition functions are executed accordingly. The Root Coordinator advances the simulation time. The simulation cycle stops when all models become passive, and

there is no external events left to process, or when the user-specified end simulation time arrives.

The following sections elaborate on the details about each subsystem.

4.2 Main Simulator

The *Main Simulator* manages the overall aspects of the simulation. It is the first object being created when the simulation starts. The class diagram of the Main Simulator is shown in the following figure.

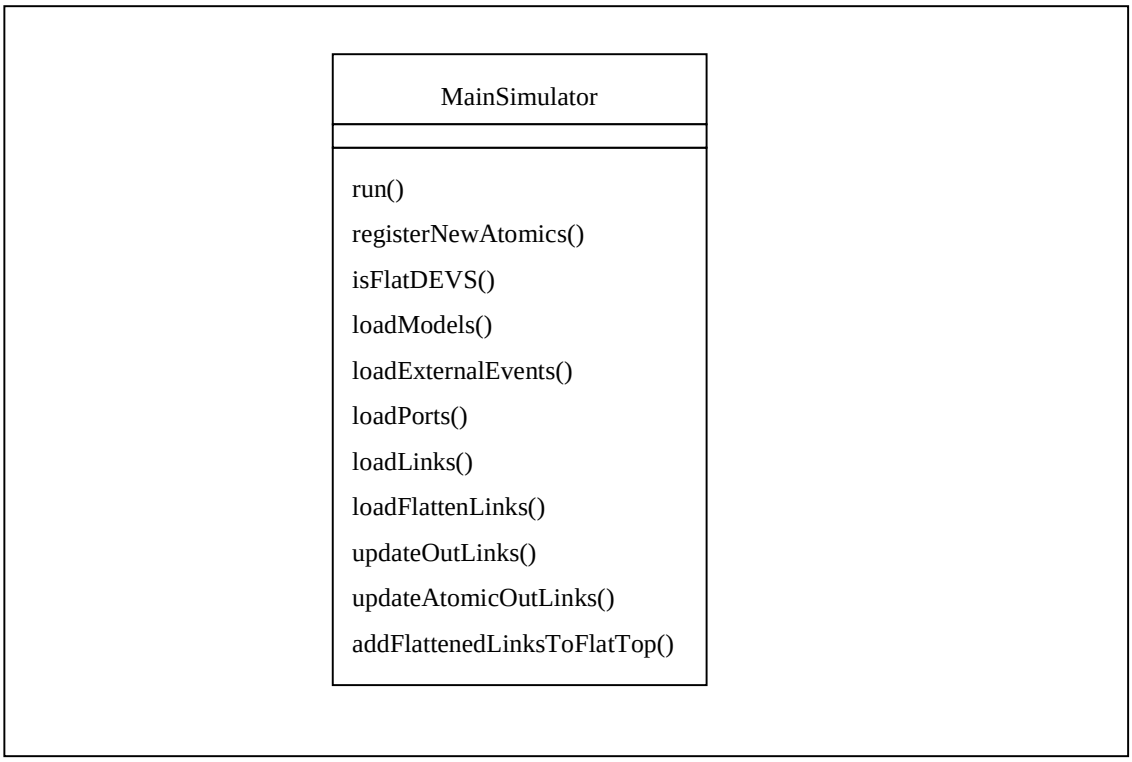

Figure 21 Main Simulator Class Diagram

The simulation cycle starts with the execution of the Main Simulator object's `run()` method. This method performs the following 4 tasks in sequence:

- *Atomic Models Registration*. The `registerNewAtomics()` method is used to store the pointers to the atomic model objects, which are C++ objects derived from the Atomic class, to Atomic Models Objects Database.

- *External Events Table Construction*. The `loadExternalEvents()` method is invoked to parse the External Events file and creates the External Events Table which is a sorted list sorted by time.
- *Models Hierarchy Tree Construction*. The methods `loadModels()`, `loadPorts()`, and `loadLinks()` are called to read in the model file and to build the Models Hierarchy Tree. If the Flattened Coordinator technique is enabled, then `loadFlattenLinks()`, `updateOutLinks()`, `updateAtomicOutLinks()`, and `addFlattenedLinksToFlatTop()` are called to construct the flattened processor hierarchy.
- *Root Coordinator Creation*. Finally, the `run()` method instantiates the Root Coordinator object and calls its `simulate()` method which sends the very first initialization message in the simulation cycle to the top component, as we will see in section 4.5.

The major portion of the Flattened Coordinator technique is implemented in the Main Simulator subsystem. The Flattened Coordinator could be either enable or disabled, and this is done by the Main Simulator's `isFlatDEVS()` method which returns a Boolean value True if the flattened coordinator is enabled, False otherwise.

As discussed in section 3.5, the Flattened Coordinator flattens the simulation hierarchy by rewiring the port links and removing the coupled components from the hierarchy. Due the absence of coordinators, however, any port links that link to coordinators' ports are re-wired to reach the far-end atomic ports. Then, the component links are handled directly by the Flattened Coordinator, which forwards the messages to simulators as needed.

The port links rewiring is implemented by the Main Simulator. As described in section 4.2, the Main Simulator calls its methods `loadModels()`, `loadPorts()`, and `loadLinks()` to construct the Models Hierarchy Tree. The method `loadLinks(Model& myModel)` builds the links defined in `myModel`, which may be either Atomic or Coupled. A *link* between two ports is a directed connection from the source port to the destination port. During the simulation, messages are sent through the

links. Usually, the destination port resides on a different component other than `myModel`. It follows, therefore, that the components in the Models Hierarchy Tree are connected by the links.

The `loadLinks()` method constructs the port links based on the DEVS specification defined by the modeller. This implies that the number of the port links and thus the volume of message passing are proportional to the level of the models hierarchy. For instance, the DEVS model structure shown in Figure 22 is a three-level hierarchy, with the atomic model `Atomic_A` and the coupled model `Coupled_A` being the first level, the coupled model `Coupled_B` being the second, and the atomic model `Atomic_B` being the third. Suppose that an external event arrives at `Atomic_A`. It will send an external message from its output port `Port_A` to the `Coupled_A's` input port `Port_B`. When the `Coupled_A` receives this message, it then sends an external message to from its input port `Port_B` to the `Coupled_B's` input port `Port_C`. Finally, `Coupled_B` sends an external message from `Port_C` to `Atomic_B's` `Port_D`, which invokes `Atomic_B's` external state transition function. In this illustration, three messages need to be generated before `Atomic_B's` external state transition is started.

By contrast, with the Flattened Coordinator, instead of calling the `loadLinks()` method, the Main Simulator calls `loadFlattenLinks()` and `updateOutLinks()` when building the Models Hierarchy Tree. The `loadFlattenLinks()` method rewires any port link that link to a coupled model directly to the far-end atomic model. For example, suppose port A links to port B, and port B links to port C, where A and C are atomic models' ports, while C is a coupled model's port. Then the `loadFlattenLinks()` method links port A directly to port C.

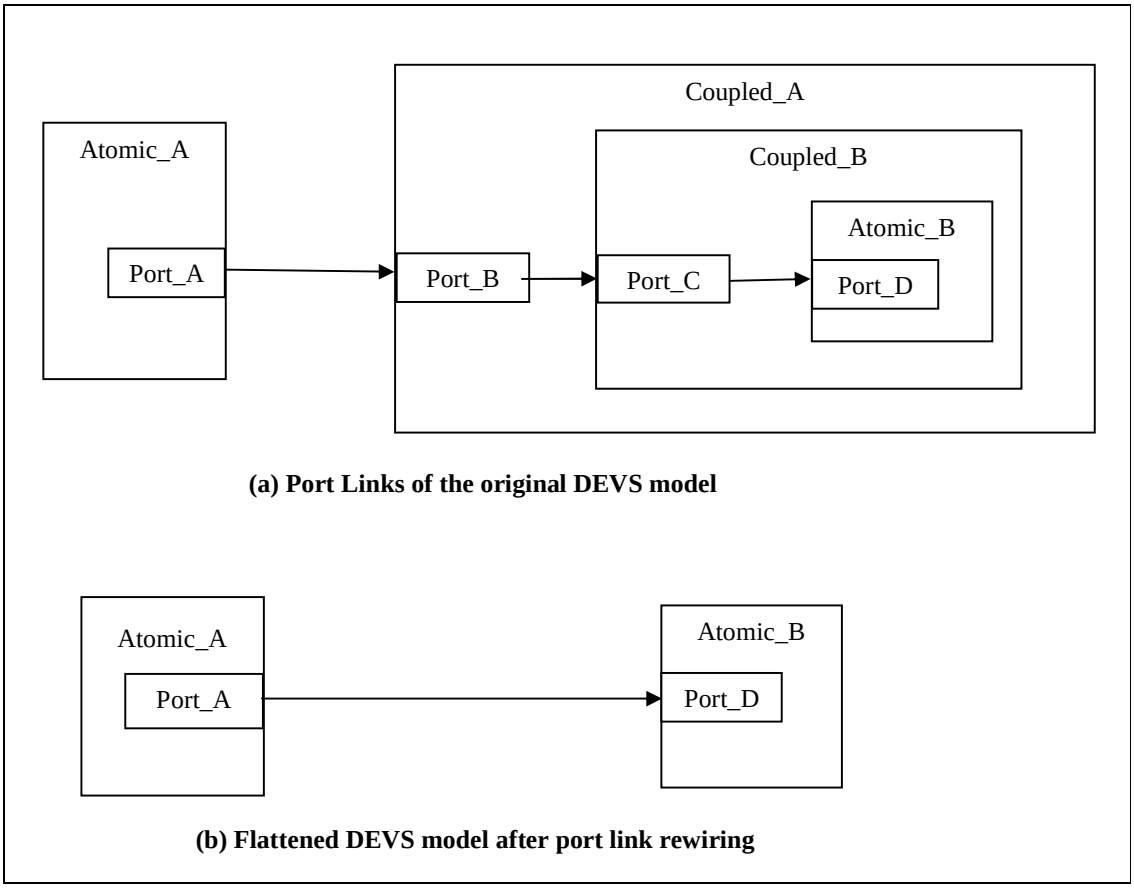

Figure 22 Port Link Rewiring by Flattened Coordinator Technique

Part B of Figure 22 shows the rewired DEVS model. Note that `Atomic_A's` input port `Port_A` is directly linked to `Atomic_B's` input port `Port_B`, and also note that the two coupled models, `Coupled_A` and `Coupled_B`, are eliminated from the hierarchy.

Similarly, the `updateOutLinks()` method also rewires any atomic model's output port that originally links to a coupled model directly to the far-end atomic model using the same algorithm used for input ports.

4.3 Modelling Subsystem

- The *DEVS Modelling Subsystem* provides a logical representation of the DEVS models defined by the modeller. The subsystem is composed by the Models Manager and the DEVS Models Hierarchy Tree.

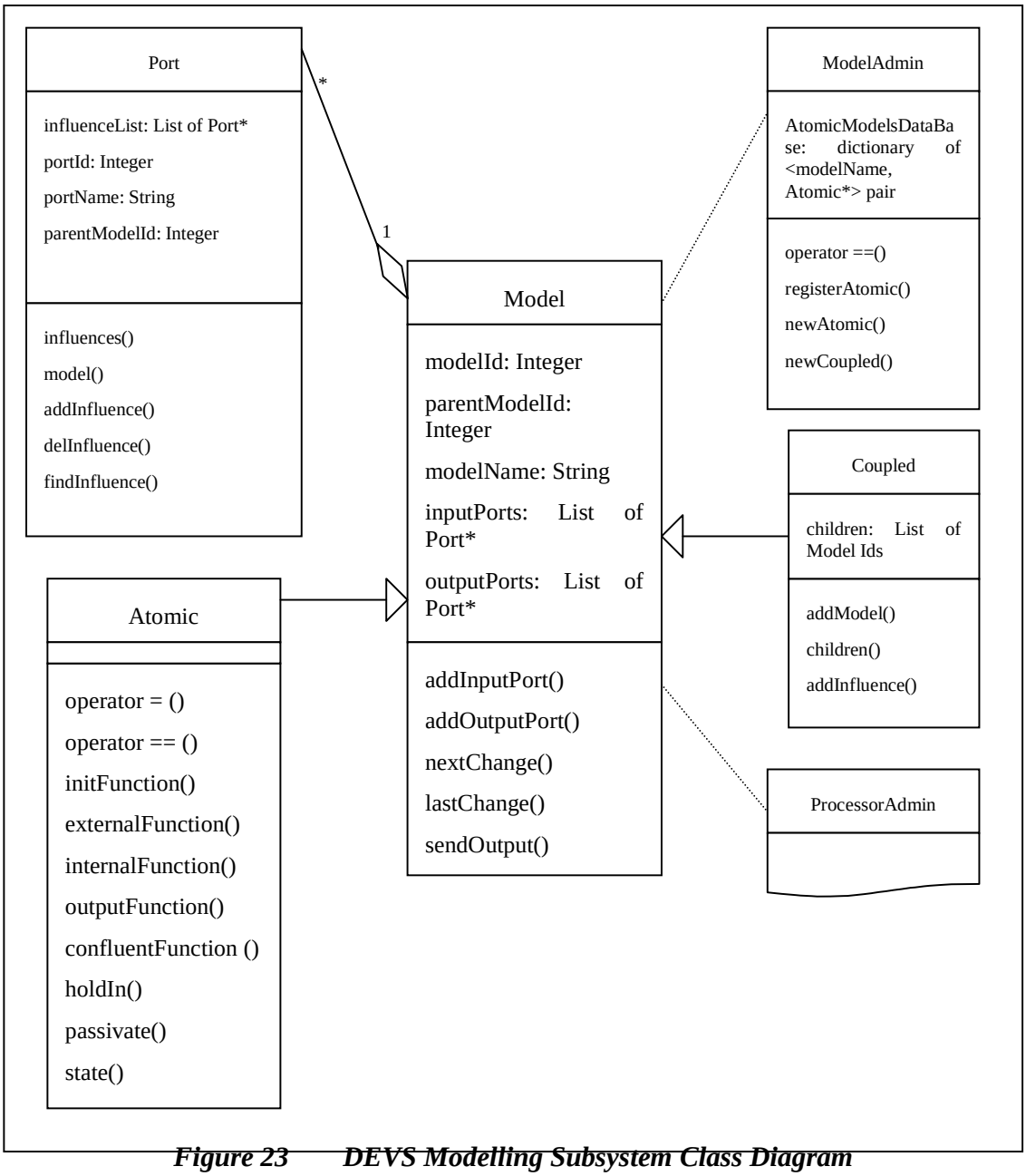

Figure 23 DEVS Modelling Subsystem Class Diagram

The *Models Manager* manages the models hierarchy. More precisely, it takes care of the following 3 tasks:

- When the Main Simulator registers the Atomic model objects, the Models Manager creates and manages the *Atomic Models Objects Database* (Refer to the class diagram in Figure 23), which is a dictionary data structure that stores the

Atomic model's string name (pointer to the "model name ↔Atomic object" pairs).

- It is employed by the Main Simulator to create the Atomic and Coupled objects in the Models Hierarchy Tree.

It employs the Processor Manager to create Processor class objects when the Main Simulator loads atomic models.

The Models Manager is implemented by the ModelAdmin class, while the implementations for the atomic and coupled models are encapsulated by the Atomic and Coupled class respectively. The *ModelAdmin object* is instantiated by the Root Coordinator. Its `registerAtomic()` method is used by the Main Simulator object's `registerNewAtomics()` method during Atomic Models Registration, and its `newAtomic()` and `newCoupled()` method are used by the Main Simulator to construct the Models Hierarchy Tree. Figure 23 represents the class diagram for the DEVS Modelling Subsystem.

Moreover, the ModelAdmin object contains the Atomic Models Objects Database that stores (modeller-defined) Atomic objects. The parent class of Atomic is the *Model* class, which is the data abstraction of the DEVS model. It also provides the data abstraction that is common to both atomic and coupled models. A Model class object has a unique `model ID`. It also contains its `parent's model ID`, so that the Models Manager can traverse the Models Hierarchy Tree upwards when necessary.

The Model also contains a list of `input ports` and a list of `output ports`. They are linked lists of pointers to Port objects. The Main Simulator's `loadModels()` method uses the `addInputPort()` and `addOutputPort()` method to construct the lists when constructing the Models Hierarchy Tree.

In addition, the Model class provides the following methods that are inherited by Atomic and Coupled class:

- `lastChange()`. Records the time of the last state transition.
- `nextChange()`. Sets the time for the next scheduled state transition.
- `sendOutput (time, port, value)`. Sends an output message through the given port.

An atomic model can be created by the modeller by including a new class derived from the *Atomic* class shown in Figure 23. In doing so, the following methods may be overloaded:

- `initFuntion()`. This method is invoked when the simulation starts. It allows one to define initial values and to execute setup functions for the model.
- `externalFunction()`. This method is invoked when an external event arrives from an input port.
- `internalFunction()`. This method is started when an internal event occurs (that is, the value of the Time Advance Function is zero).
- `outputFunction()`. This method executes before the internal function in order to generate outputs for the model.
- `confluentFunction()`. This method is invoked when an external event and an internal event takes place simultaneously. This function is an important feature offered by the P-DEVS formalism. The function enables the modeller to control the collision behaviour.

These functions are equivalent to those defined in the formal specifications for atomic models. In addition the following primitives can be used when defining an atomic model.

- `holdIn (state, time)`. A model executing this method remains in *state* during *time (ta(s) == time)*. When the time is consumed (i.e., ta(s) == 0), the model executes the internal transitions. This method was included to make the definition of the duration function easy.
- `passivate()`. The model enters in passive mode (i.e., phase == passive; ta(s) == infinite) and it is only reactivated by an external event.
- `state()`. Returns the present model phase.

Figure 23 also shows that the *Coupled* class object contains a `list of its children's model IDs`. The list is constructed by the `addModel()` method. It defines the containment relation between the coupled component and its children. The port connections that link these children are created by the `addInfluence()` method, which is employed the by Main Simulator's `loadLinks()` method during the Models Hierarchy Tree construction.

As mentioned earlier, a Model may contain zero or more input ports and output ports. The *Port* objects are created by the Main Simulator's `loadPorts()` method during the models hierarchy construction time. A Port object contains a numerical ID and a name. It also stores its parent model's ID, which can be retrieved by its `model()` method. This implies that the port's parent component must be created prior to the creation of the port, and this order is enforced by the Main Simulator's models hierarchy construction algorithm.

Once a Port is created, the Main Simulator starts to build the Port's `influence list`, which is a list of pointers to a set of Port objects representing the port's destination end. The Port's parent coupled component's `addInfluence()` method calls the Port's `addInfluence()` method to build the list.

4.4 GGAD Model Loader

E-CD++ incorporates a GGAD model loader that parses GGAD files and builds equivalent atomic models. The GGAD model loader is part of the Modelling subsystem. It consists of the following software modules:

- GGAD Parser
- Symbol Table
- Syntax Tree
- GGAD Transitions Execution Engine
- Atomic Model Adapter

The GGAD Model Loader's design follows the classic compiler design pattern that groups these modules into a front-end and a back-end. The front-end contains the parser, symbol table and syntax tree. The GGAD parser is written in lex and yacc. It parses the input GGAD files and builds the syntax tree and the symbol table in a similar way as what a typical compiler's front-end would do.

The back-end consists of the GGAD Transitions Execution Engine and the Atomic Model Adapter. The former interacts with the syntax tree and the symbol table to carry out the state transitions, while the latter makes GGAD models behave exactly the same as if they were derived Atomic classes written in C++. This is done by the Atomic Model Adaptor providing the same API as that provided by the Atomic class. Providing a consistent API makes the integration of the GGAD model loader with the rest of the E-CD++ code become easy.

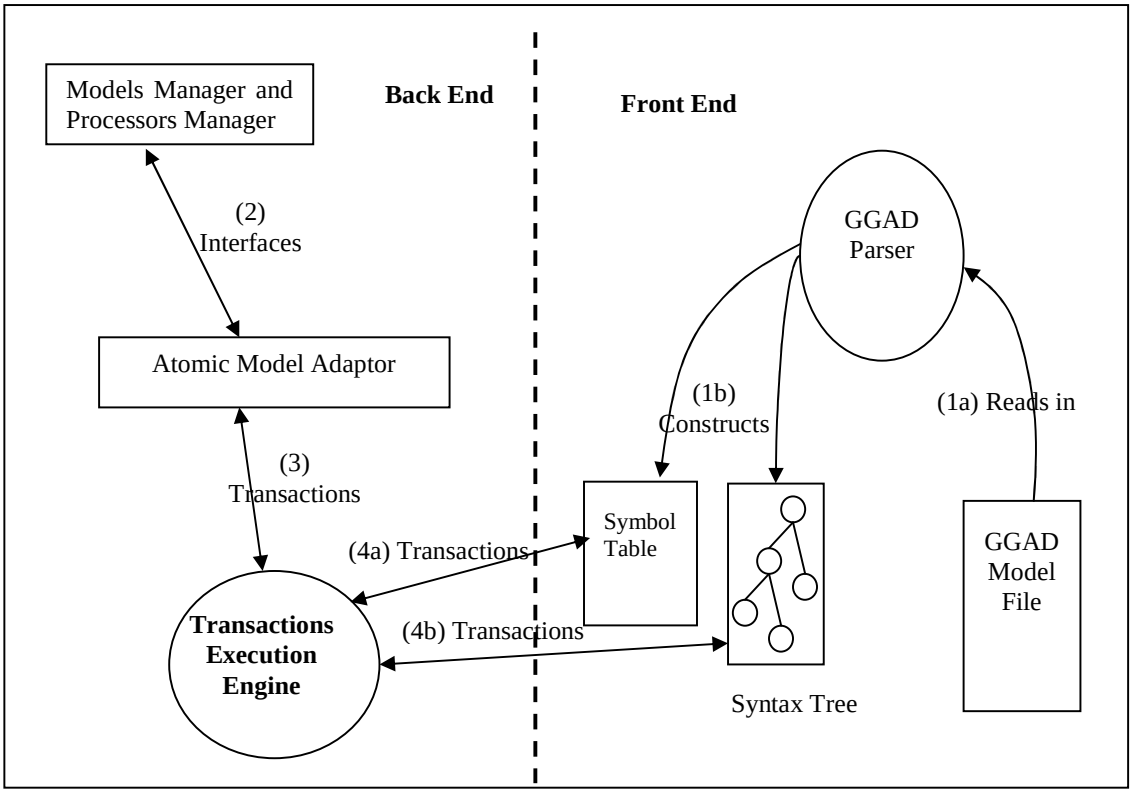

Figure 24 GGAD Model Loader Architectural Overview

The interactions and relations among these modules are illustrated in Figure 24, and the design walk-through of the GGAD parsing process is explained as follows:

1. Before the simulation cycle begins, the GGAD parser reads in the GGAD model file (1a) and constructs the symbol table and the syntax tree (1b).
2. When the simulation cycle begins, the Models Manager and the Processors Manager execute atomic models' state transition functions at the scheduled time. The Models Manager calls the transition function APIs provided by GGAD's Atomic Model Adaptor. These APIs are consistent with that defined in the `Model` class, so that it does not require any special handling for GGAD models.
3. The Atomic Model Adaptor calls the proper GGAD Transitions Execution Engine APIs in order to fulfil the state transition requests obtained from the simulation subsystem.
4. The Transitions Execution Engine starts to execute. It interacts with the symbol table (4a) and the syntax tree (4b) and carries out the state transitions. (The Transitions Execution Engine is quite complex. Its design is explained in section 4.4.4.)

4.4.1 GGAD Parser

The *GGAD Parser* is written in Lex and Yacc, which are the tools used to define the context-free grammar of GGAD model files. The GGAD Parser parses the GGAD model file and builds the syntax tree and the symbol table. The implementation of the GGAD Parser is encapsulated in the `GgadParser` class. The class diagram is shown in Figure 25.

The GgadParser object contains the symbol table (`ggadSymTbl`) and the Transitions Execution Engine (`ggadTransEngine`). Its `parse()` method serves as the main body of the parser, which performs two tasks in sequence: it first calls the `initSymbolTable()` method to initialize the symbol table and then adds to it the GGAD keywords, which are summarized in Table 1 below.

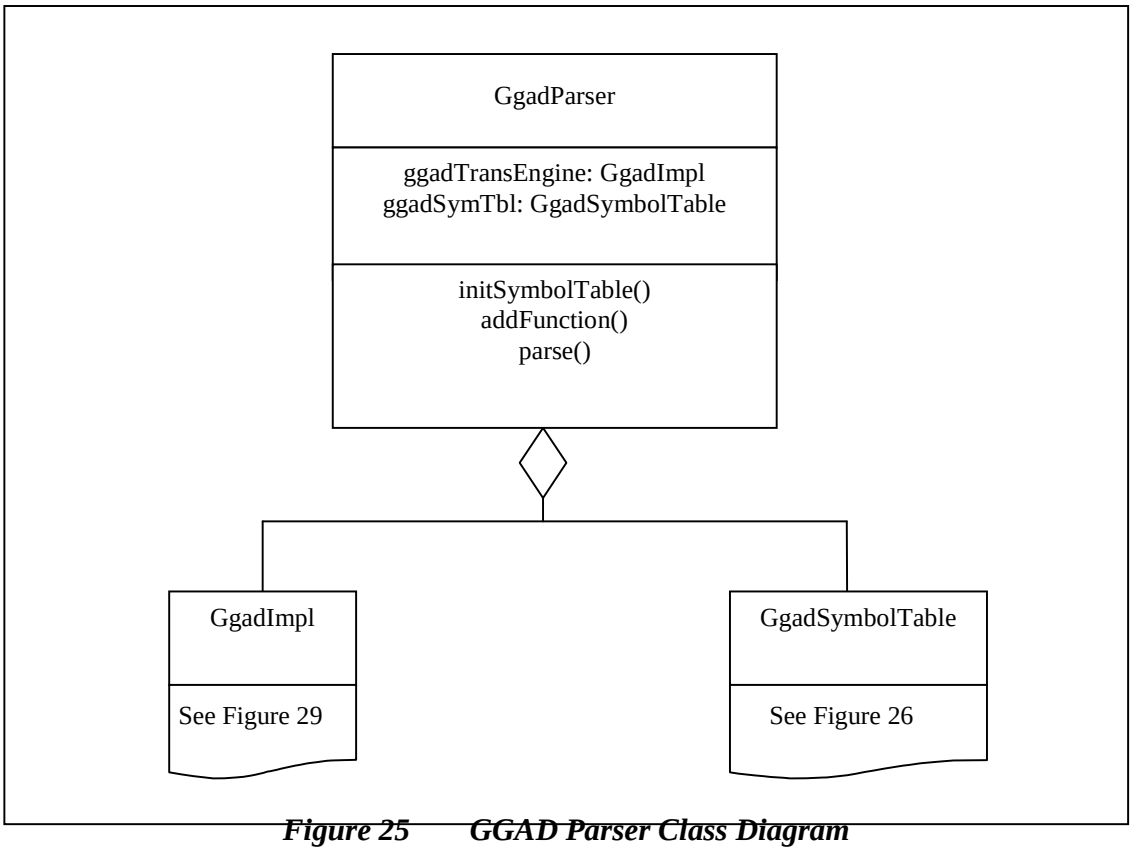

Figure 25 GGAD Parser Class Diagram

Keyword	Description	Example
in	input port list	**in**: in_port1 in_port2 …
out	output port list	**out**: out_port1 out_port2 …
state	atomic model state list	**state**: state1 state2 …
initial	initial state	**initial**: state1
int	Internal transition function	**int**: state1 state2 out_port ! output_value { ggad_transition_statements … }
ext	External transition function	**ext**: state1 state2 in_port ? input_value { ggad_transition_statements … }
infinite	infinite elapse time	some_idle_state: **infinite**
var	local variables list	**var**: variable1 variable2 …
pi	Constant Pi	my_variable = **pi**;

Table 1 GGAD Keywords

Since these keywords have special meanings in GGAD, they cannot be used for any other purposes. That's why they are saved in the symbol table before the input file is parsed, so

that when the parser encounters variable names, port names or state names during parsing, it checks them against the keywords in the symbol table. If any of the keywords are used as these names, the parser will raise an error. Furthermore, as part of the Symbol Table initialization, the `registerFunctions()` method is called by the constructor of the `ggadSymTbl` class to add GGAD built-in functions into the symbol table.

The second task that the `parse()` method performs is to call the `GGADparse()` routine to start parsing process. `GGADparse()` is generated by the lex and yacc based on the GGAD grammar, which is defined in Appendix B. It parses the GGAD model file and creates the syntax tree. It also adds more symbols to the symbol table.

4.4.2 GGAD Symbol Table

The *GGAD Symbol Table* stores input and output port names, state names, variable names, keywords, and built-in functions. It is implemented by the `GgadSymbolTable` class. The class has two data members given below:

- `ggadSymbols` is a dictionary of symbols, such as port names, state names, variable names, and keywords listed in Table 1. The symbols in the dictionary are represented by the `GgadSymbol` class and can be searched by their symbol names. The symbols can be added to the dictionary by the `addSymbol()` and `setSymbolType()` methods, and can be searched and retrieved by the `getSymbolType()`method.

- The second data member, namely `ggadFunctionTable`, is a built-in function table. It is a dictionary data structure of <function_name, function pointer> pairs. That is, it a list of pointers to `GgadFunc` objects that are indexed by function names. The built-in functions can be searched and retrieved by `getFunctionByName()` method.

Figure 26 is the class diagram for `GgadSymbolTable`, `GgadSymbol`, and `GgadFunc`.

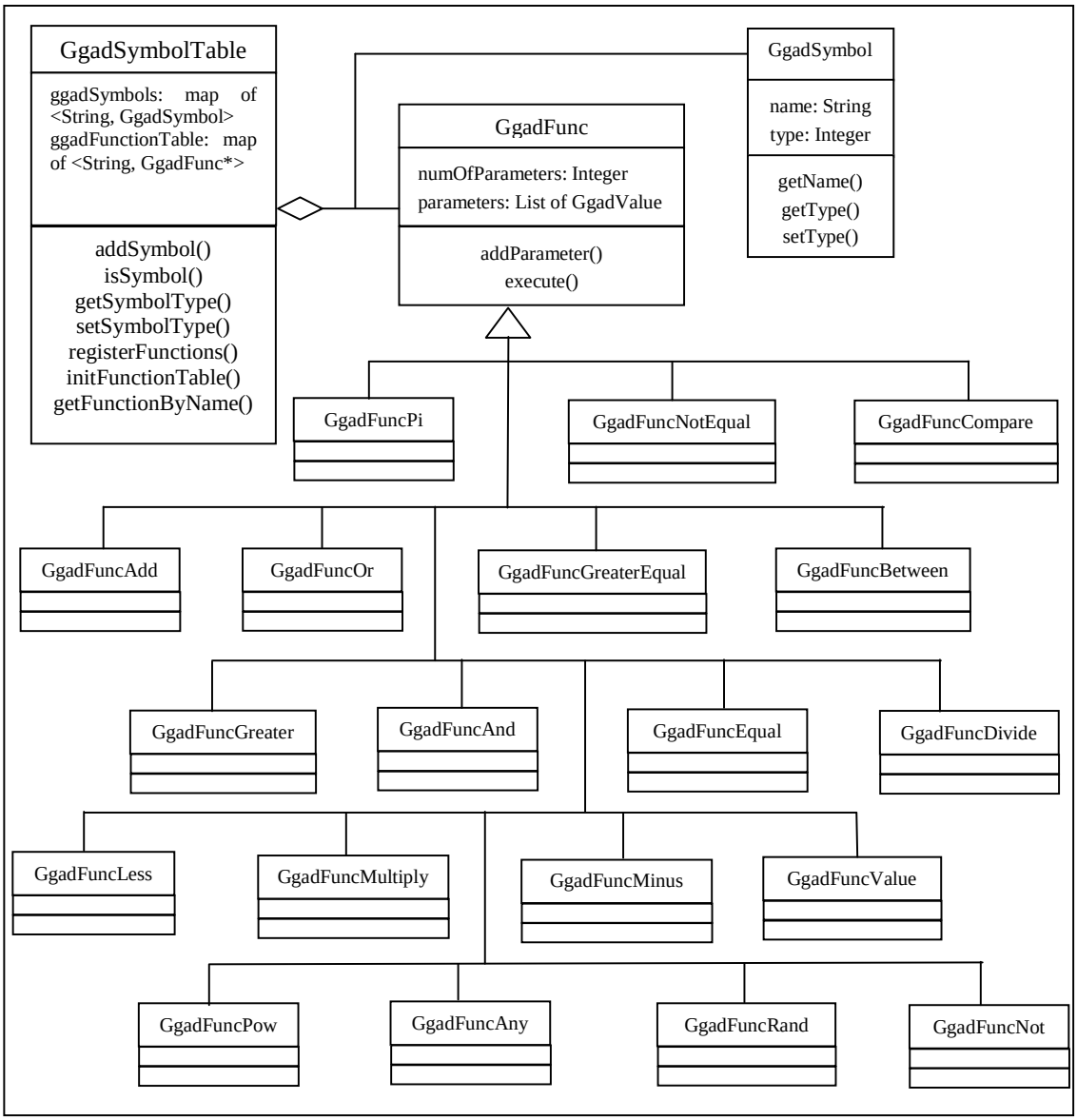

Figure 26 GGAD Symbol Table Class Diagram

The `GgadSymbol` class is a simple class that contains the `symbol name` and its associated `symbol type`. The symbol type determines if the symbol is a port, state, variable, or a keyword. The class constructor takes in the symbol name as its input parameter, so that the symbol name is stored when the `GgadSymbol` object is

instantiated. Its symbol type, however, is set via the `setType()` method. The symbol name and symbol type can be retrieved by the `getName()` and `getType()` method.

Function	Class Name	Parameters	Description of execute() method
value	GgadFuncValue	my_var	Returns the value of variable my_var
add	GgadFuncAdd	a, b	Returns a + b
minus	GgadFuncMinus	a, b	Returns a – b
multiply	GgadFuncMultiply	a, b	Returns a * b
divide	GgadFuncDivide	a, b	Returns a / b
pow	GgadFuncPow	a, b	Returns a to the power b
between	GgadFuncBetween	a, b, c	Returns 1 if a <= b <= c, 0 if not
compare	GgadFuncCompare	a, b, c, d, e	If a < b, return c; If a == b, return d; else return e
equal	GgadFuncEqual	a, b	Returns 1 if a == b 0 if not
notequal	GgadFuncNotEqual	a, b	Returns 1 if a != b, 0 if not
less	GgadFuncLess	a, b	Returns 1 if a < b, 0 if not
greater	GgadFuncGreater	a, b	Returns 1 if a > b, 0 if not
greaterequal	GgadFuncGreaterEqual	a, b	Returns 1 if a >= b, 0 if not
and	GgadFuncAnd	a, b	Returns (a && b)
or	GgadFuncOr	a, b	Returns (a || b)
not	GgadFuncNot	a	Returns !a
any	GgadFuncAny	my_port	Returns 1 if the input port my_port receives any input values, 0 if not
rand	GgadFuncRand	a, b	Returns a random number in range [a, b]
pi	GgadFuncPi	N/A	Returns 3.14159

Table 2 GGAD Built-in Functions (note: parameters a, b, c, d and e have double data type; my_var and my_port are strings)

The first column in Table 2 are the function names. The 2nd and 3rd column are the associated class name and the parameter list respectively, while the last column describes the implementation of the `execute()` method.

The GGAD language specification defines 19 built-in functions. GGAD users can use these functions to implement their state transition functions. The symbol table's function table (`ggadFunctionTable`) contains the behavioural specifications of these functions. And the function table entries are created by the method `GgadSymbolTable::registerFunctions()`, which is invoked by the class constructor. Therefore, these built-in functions are all available for execution once the symbol table is constructed. This approach simplifies the GGAD parser's design, which would otherwise have to dynamically load the needed built-in functions.

Every built-in function table entry contains a pointer to a `GgadFunc` object. The `GgadFunc` class provides a data abstraction of a GGAD function. It contains two data members: the list of function parameters and the total number of the parameters. The functions parameters are added by the `addParameter()` method. The `execute()` method is a pure virtual function that is overloaded by its subclasses.

The `GgadFunc` class has 19 subclasses, each of which specifies one particular built-in function's behaviour. These subclasses do not introduce any new data members or methods. Their behaviours are differentiated by their implementations of the `execute()` method. This method operates on the function parameters and returns the function's result. Different functions have different implementations. Table 2 illustrates the various implementations of the `execute()` method.

4.4.3 GGAD Syntax Tree

The *Syntax Tree* is used by the GGAD Transitions Execution Engine to carry out the model simulation. It is a tree structure of GgadSyntaxNode class objects. The GgadSyntaxNode class has 6 subclasses (refer to Figure 27 for the class diagram):

- `GgadFunctionNode`
- `GgadConstantNode`
- `GgadInputNode`
- `GgadPortInNode`

- `GgadVariableNode`
- `GgadActionNode`

In addition, the `GgadActionNode` class has 3 subclasses:

- `GgadActionListNode`
- `GgadSetVariableNode`
- `GgadNullActionNode`

These classes, among with those representing the symbol table, form a run-time presentation of the GGAD language schematics. In other words, the symbol table and the syntax tree provide the behaviours of an atomic model in terms of C++ objects that can be executed during run time. As explained in the previous section, the symbol table is mainly used to store the input and output port names, state names, local variable names, keywords, and built-in functions. The syntax tree, on the other hand, mainly represents the internal and external transition functions, whose context-free grammar is defined in Figure 28.

When parsing the input GGAD model file, the GGAD Parser translates the elements of the atomic model, such as input and output ports, states, variables, and state transition functions, into symbol table entries and GGAD syntax node objects. Details of this translation process are given below:

- When parsing the internal and external transition functions from the GGAD model file, the parser applies the grammar rules in a recursively descendent order. That is, it starts with applying rule 1 or rule 2, depending on the transition type, and recursively break the non-terminals into other rules. The non-terminal “Actions” in rule 1 and 2 can be broken down into either a list of “Actions” separated by semicolons (rule 9 & 12) including a null action (rule 10). For the null action case, the parser simply creates a `GgadNullActionNode` object, which has no data members or methods. In any other cases, the “ActionList” may contain one or more “Actions” (rule 11 and 12); so, the parser creates a

`GgadActionListNode` object, which contains a data member, "`actions`", and a method, `addAction()`. The "`actions`" is a list of pointers to `GgadSyntaxNode`, and the parser calls `addAction()` to insert action objects to the list.

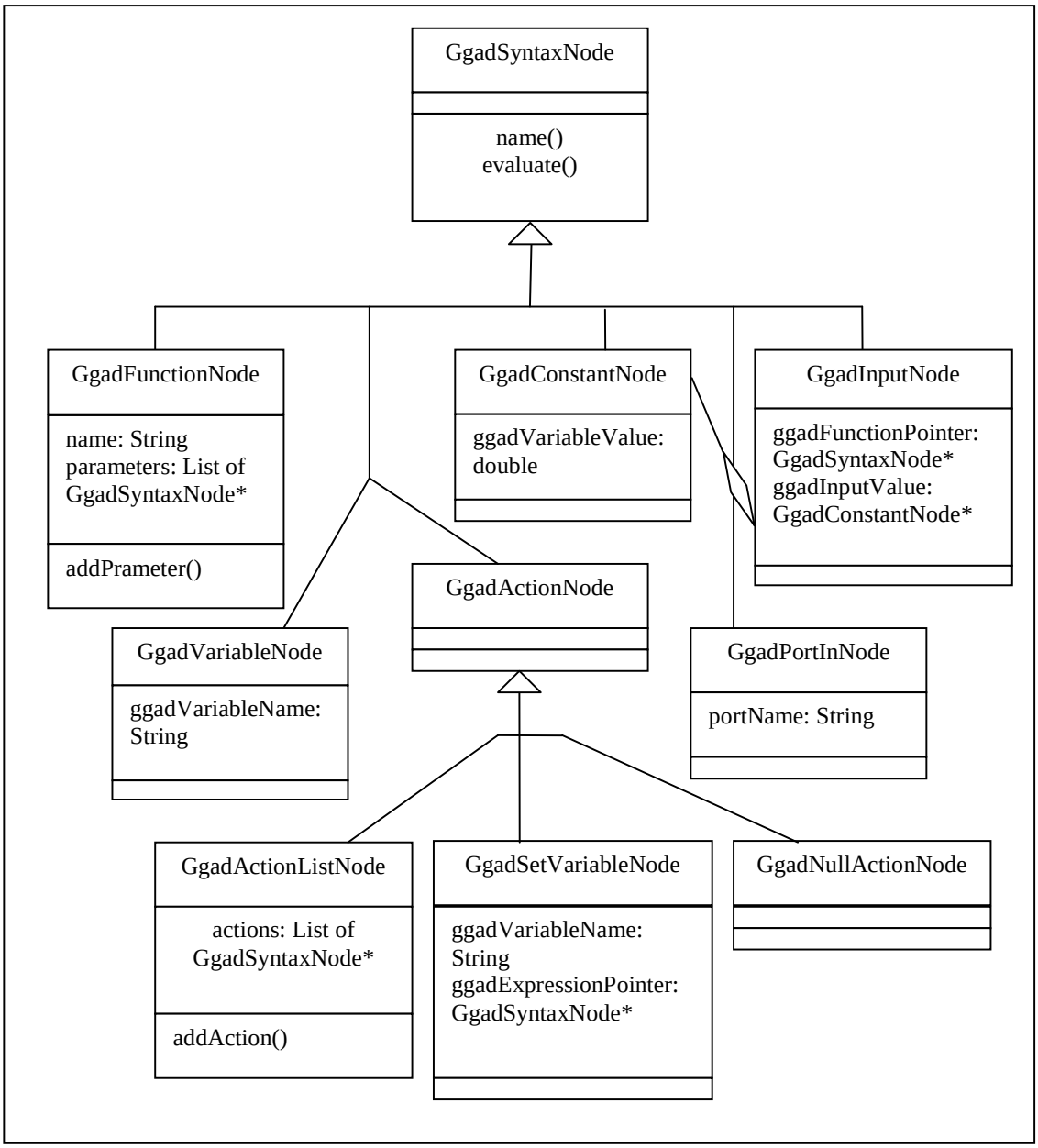

Figure 27 GGAD Syntax Tree Class Diagram

- The non-terminal “Action” (rule 13) defines the syntax of local variables’ assignments. The parser creates a `GgadSetVariableNode` object when applying this rule. The `GgadSetVariableNode` object contains two data members: `ggadVariableName` and `ggadExpressionPointer`. The former is a string representing the local variable name, while the latter is a pointer to `GgadSyntaxNode` representing the variable value, which is an “Expression”.

- An “Expression” can be break down into a GGAD built-in function call (rule 5), a port name (rule 6), a variable name (rule 7) or a numerical constant (rule 8). For a function call (rule5), the parser creates a `GgadFunctionNode` object. This object contains the `function_name` and a list of `function parameters` as its data members. Rule 17 –19 define the syntax of function parameters. When applying one of these rules, the parser calls `GgadFunctionNode`’s `addParameter()` method to add the parameter to the object’s parameter list.

- If the “Expression” is a port name (rule 6), the parser creates a `GgadPortInNode` object, which has a string type data member `port_name`. Similarly, the parser creates a `GgadVariableNode` or a `GgadConstantNode` object if the “Expression” is a variable name (rule 7) or a constant (rule 8).

- The last type of syntax node we need to introduce is `GgadInputNode`. This syntax node is created when the parser reads in the input value from an input port defined in an external transition (rule 2). The `GgadInputNode` has two data members: The `ggadFuntionPointer` data member is a pointer to `GgadSyntaxNode`, which represents the “Expression” in rule 2. This “Expression” is applied upon the input port with the condition that triggers the external transition to occur. For example, the expression: “any (myPort)” means that if the input port “myPort” has any incoming data arrived, start the external

transition. The other data member, `ggadInputValue`, is a pointer to `GgadConstantNode`. It represents the incoming value at the port.

```
1)   IntDef -> 'int' ':' STATE STATE PortValueOutList Actions
2)   ExtDef -> 'ext' ':' STATE STATE Expression '?' CONSTANT Actions
3)   PortValueOutList -> PORT '!' Expression PortValueOutList
4)   PortValueOutList -> /* empty */
5)   Expression -> FunctionCall
6)   Expression -> PORT
7)   Expression -> VARIABLE_NAME
8)   Expression -> CONSTANT
9)   Actions -> '{' ActionList '}'
10)  Actions -> /* empty */
11)  ActionList -> Action ';'
12)  ActionList -> ActionList Action ';'
13)  Action -> VARIABLE_NAME '=' Expression
14)  FunctionCall -> FUNCTION_NAME '(' ParameterList ')'
15)  ParameterList -> Parameter
16)  ParameterList -> Parameter ',' ParameterList
17)  Parameter -> CONSTANT
18)  Parameter -> VARIABLE_NAME
19)  Parameter -> PORT
```

Figure 28 Context grammar of GGAD internal and external transition functions

Figure 27 shows that GgadSyntaxNode class has two virtual methods, name() and evaluate(), that can be overloaded by its subclasses. The name() method simply returns the class name as a literal string (mainly for debugging purposes). The evaluate() method, however, carries out syntax nodes activities. The different types of the syntax nodes, introduced in Figure 27, have different implementations (which are described in Table 3).

Note that it is our design intention to make `GgadSyntaxNode::evaluate()`a pure virtual function, so that its implementation in the subclasses can vary. This design has two advantages:

- It makes the Syntax Tree is scalable and easy to expand. So, the GGAD language evolution becomes relatively easy. Suppose, for example, in the future we want to add confluent functions to GGAD. We only need to add a new subclass to

`GgadSyntaxNode` class and implement the behaviour of confluent functions in its `evaluate()` method. The rest of the backend code need not to be changed.

- It makes the interface between the Transitions Execution Engine and the Syntax Tree very simple. This design takes the advantage of polymorphism, so that the Transitions Execution Engine only need to call the abstraction method `GgadSyntaxNode::evaluate()` to execute various transition actions, as opposed to finding different syntax node types and calling different APIs.

Class	Description of the evaluate() method
GgadFunctionNode	1. Calls the mothod GgadSymbolTable::getFunctionByName() to retrieve the GGAD function (GgadFunc object) from the symbol table. 2. Uses GgadFunc's addParameter() method to add parameters to the function object 3. Calls function object's execute() method (see Table 2) to execute the function and return the result
GgadConstantNode	Returns the data member ggadVariableValue, which stores the constant value
GgadInputNode	1. Calls ggadFunctionPointer->evaluate(). 2. Calls ggadInputVaue->evaluate() 3. Returns 1 if return values from step 1 & 2 are the same; 0 otherwise
GgadPortInNode	Returns the data member portName, which is an input port name
GgadVariableNode	Returns the value of the variable whose name is saved in the data member ggadVariableName
GgadActionNode	Nil operation
GgadActionListNode	Calls the `evaluate()` method of every syntax node in the data member `actions`
GgadSetVariableNode	Calls the evaluate() method of the syntax node pointed by ggadExpressionPointer. Then it assigns the value to the variable with the name saved in ggadVariableName
GgadNullActionNode	Nil operation

Table 3 Behaviours of the evaluate() method in GGAD syntax node classes

More details on the Transitions Execution Engine follow in the following section.

4.4.4 GGAD Transitions Execution Engine

The *GGAD Transitions Execution Engine* is the main body of the GGAD Model Loader. It is responsible for the executions of external, internal, output, and initialization transition functions of the GGAD models. It interacts with the symbol table and with the GGAD syntax nodes objects to carry out the GGAD simulation activities. The implementation of the execution engine is encapsulated in the `GgadImpl` class, of which the class diagram is shown in Figure 29.

Figure 29 shows that the `GgadImpl` class contains a collection of data members that form a data portfolio for the GGAD model:

- First of all, the data member `symTable` is a pointer to the symbol table, so that the execution engine has direct access to it (via its `setSymbolTable()` and `getSymbolTable()` methods).

- The `GgadImpl` class contains a list of GGAD states (`ggadStates`), which is constructed by the parser. When parsing a state from the GGAD file, the parser calls the methods `GgadImpl::addState()` and `GgadImpl::setTimeAdvance()` to add the state and set its duration.

  The GGAD state is represented by the `GgadState` class, which contains the name of the state and its duration as the data members. The state information can be retrieved by the method `GgadImpl::getState()`.

- `GgadImpl` also has a list of variables (`ggadVariables`) as the data member. This list is also built by the parser calling the `addVariable()` method during parsing time. The variables in the list can be retrieved by the method `getVariable()`.

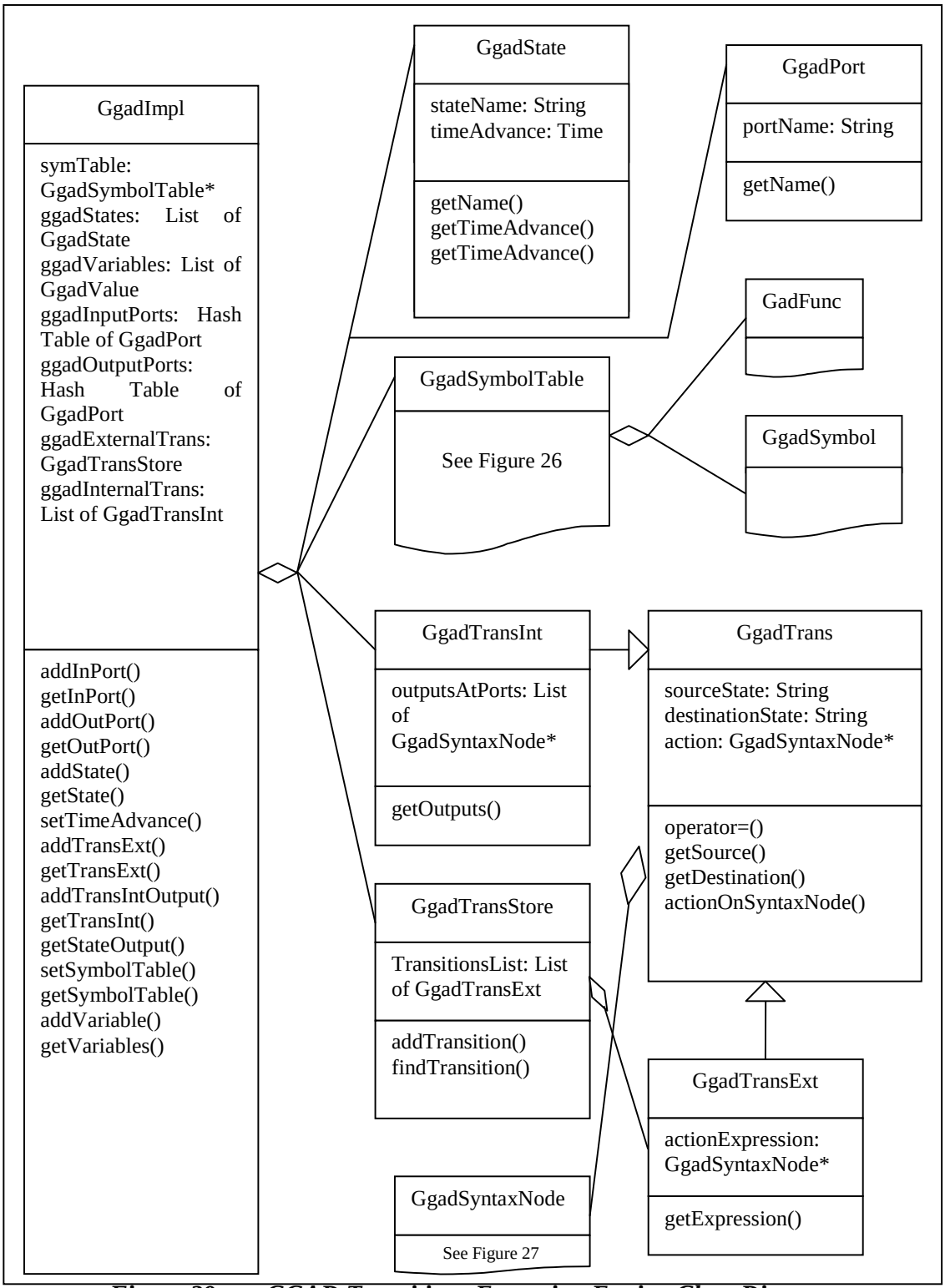

Figure 29 GGAD Transitions Execution Engine Class Diagram

- The GGAD model's input ports and output ports are represented by `GgadImpl::ggadInputPorts` and `GgadImpl::ggadOutputPorts`, respectively. Both of the data members are hashing tables of `GgadPort` objects. The `GgadPort` class contains the port name as its data member. Both lists are built by the parser, which calls `addInPort()` and `addOutPort()` to add input and output ports when parsing the GGAD file. The elements in the two lists can be retrieved by the methods `GgadImpl::getInPort()` and `GgadImpl::getOutPort()`.

- Finally, the data members `ggadInternalTrans` and `ggadExternalTrans` represent internal and external state transitions, respectively. A *state transition* can be specified by three attributes: source state, destination state, and state transition function. These attributes are represented in the `GgadTrans` class as its data members. The method `GgadTrans::actionOnSyntaxNode()` carries out the state transition by updating the model's state from the source state to destination state and also by executing the state transition function. The transition function is represented by the data member `action`, which is a pointer to the Syntax Tree. The `actionOnSyntaxNode()` method calls the `evaluate()` method (see Table 3) in the Syntax Tree which executes the state transition actions defined in the GGAD file.

  The data member `ggadInternalTrans` is a list of `GgadTransInt`, which represents the internal state transition and the output function. It is a list because GGAD allows multiple definitions of internal state transitions, and the list is constructed by `GgadImpl::addTransIntOutput()`, which is called by the parser to add the internal transition function and the output function. `GgadTransInt` is a subclass of `GgadTrans`. Its method `actionOnSyntaxNode()` executes the internal transition function, while its `getOutPuts()` method runs the output function.

Similarly, the data member `ggadExternalTrans` is a list of `GgadTransExt`, which represents the external state transition. This list is also created by the parser calling `GgadImpl::addTransExt()`to add each individual external transition. The `GgadTransExt` class is a subclass of `GgadTrans`. Its `actionOnSyntaxNode()` method executes the external transition function. It also contains a data member called `actionExpression`, which is a pointer to the Syntax Tree. This data member represents the "Expression" in rule 2 of the GGAD grammar in Figure 28, and the `getExpression()` method executes the expression by calling `actionExpression`'s `evaluate()` method.

The Transition Execution Engine is used by the Atomic Model Adaptor. The Adaptor calls the member methods in the `GgadImpl` class to carry out state transitions. Details about the adaptor are discussed in the following section.

4.4.5 GGAD Atomic Model Adaptor

The `Ggad` class serves as the *Atomic Model Adaptor* -- a software adaptation layer that encapsulates the GGAD Transitions Execution Engine, as described in Figure 30.

The `Ggad` class is a subclass of the `Atomic` class. Thus, it can provide the same public methods as that provided by `Atomic`:

- Model& Ggad::initFunction ()
- Model& Ggad::externalFunction (const ExternalMessage &)
- Model& Ggad::internalFunction (const InternalMessage &)
- Model& Ggad::outputFunction (const InternalMessage &)

It is our design intention to make the Models Manager and the Processor Manager from the Modelling Subsystem only interface with these APIs provided by the `Ggad` class, so that the detailed implementation of GGAD is hidden away from other subsystems. This

decoupling makes implementation changes to GGAD easy, since they will not impact other subsystems.

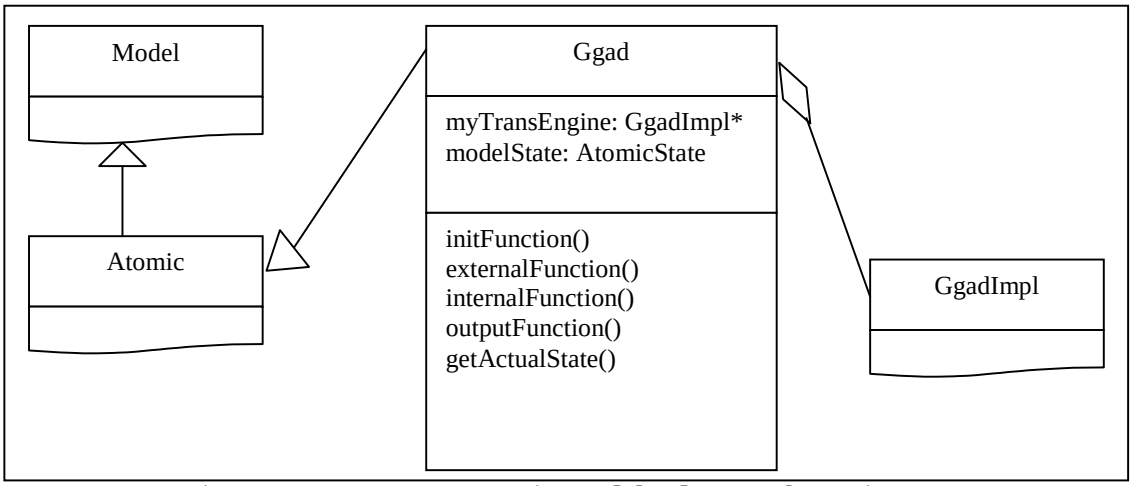

Figure 30 GGAD Atomic Model Adaptor Class Diagram

On the implementation side, the `Ggad` class needs to map its public methods to the methods in the Transitions Execution Engine. It contains a pointer to the Transitions Execution Engine as its data member, namely `myTransEngine`, so that it executes the public methods by invoking the appropriate Execution Engine methods through `myTransEngine`. Table 4 summarizes how the `Ggad` methods employ the methods in the `GgadImpl` class. Those `GgadImpl` methods are explained in the previous section.

Ggad method name	**Implementation of the Ggad method**
Ggad::initFunction()	Calls `myTransEngine->getState()` to find the initial state and sets `modelState` to initial state
Ggad::externalFunction()	Calls `myTransEngine->getTransExt()` to execute the external state transition function
Ggad::internalFunction()	Calls `myTransEngine->getTransInt()` to execute the internal state transition function
Ggad::outputFunction()	Calls `myTransEngine->getStateOutput()` to execute the output function

Table 4 Implementations of the Ggad class methods

With the design of this adaptation layer, therefore, the entire GGAD subsystem can be seamlessly integrated into E-CD++.

4.5 Simulation Subsystem

The *Simulation Subsystem* consists of Simulators, Coordinators, and the Processors Manager. Figure 31 shows the Simulation subsystem.

The *Processor* class is the parent class for Simulator and Coordinator. So, it abstracts the commonality between them. This class contains the following data members:

- Each `Processor` has a unique `Processor_ID`, which is an integer. This ID is used by the Processor Manager (i.e., `ProcessorAdmin` class) to keep track of each Simulator and Coordinator.
- The `Processor` class uses two data members to keep the time of the simulation:

  t_L *Absolute time of last transition*
  t_N *Time of next transition relative to* t_L

  The `lastChange()` and `nextChange()` method returns t_L and t_N, respectively. The absoluteNext() method returns the absolute time of the next transition, which is the sum of t_L and t_N.
- The `Processor` class also contains a data member called `externalMsgs`, which serves as the *message bag* of external messages. The message bag is a device introduced by the P-DEVS formalism to achieve parallel simulation. Its functionality is implemented by the *MessageBag* class, which contains a list of pointers to the `Message` objects. (The Message class will be explained in the next section.) In addition, its `addExternalMessage()` and `eraseAll()` method are used by Processors to insert individual external messages to the bag and empty the entire bag, respectively.

- The last data member need to mention is `model`, which is an instance of the `Model` class. Each `Processor` object contains one `Model` object, which reflects the one-to-one mapping relation between processors hierarchy and models hierarchy. Furthermore, the Processor's message handlers can access the model's transition functions and port links through this data member.

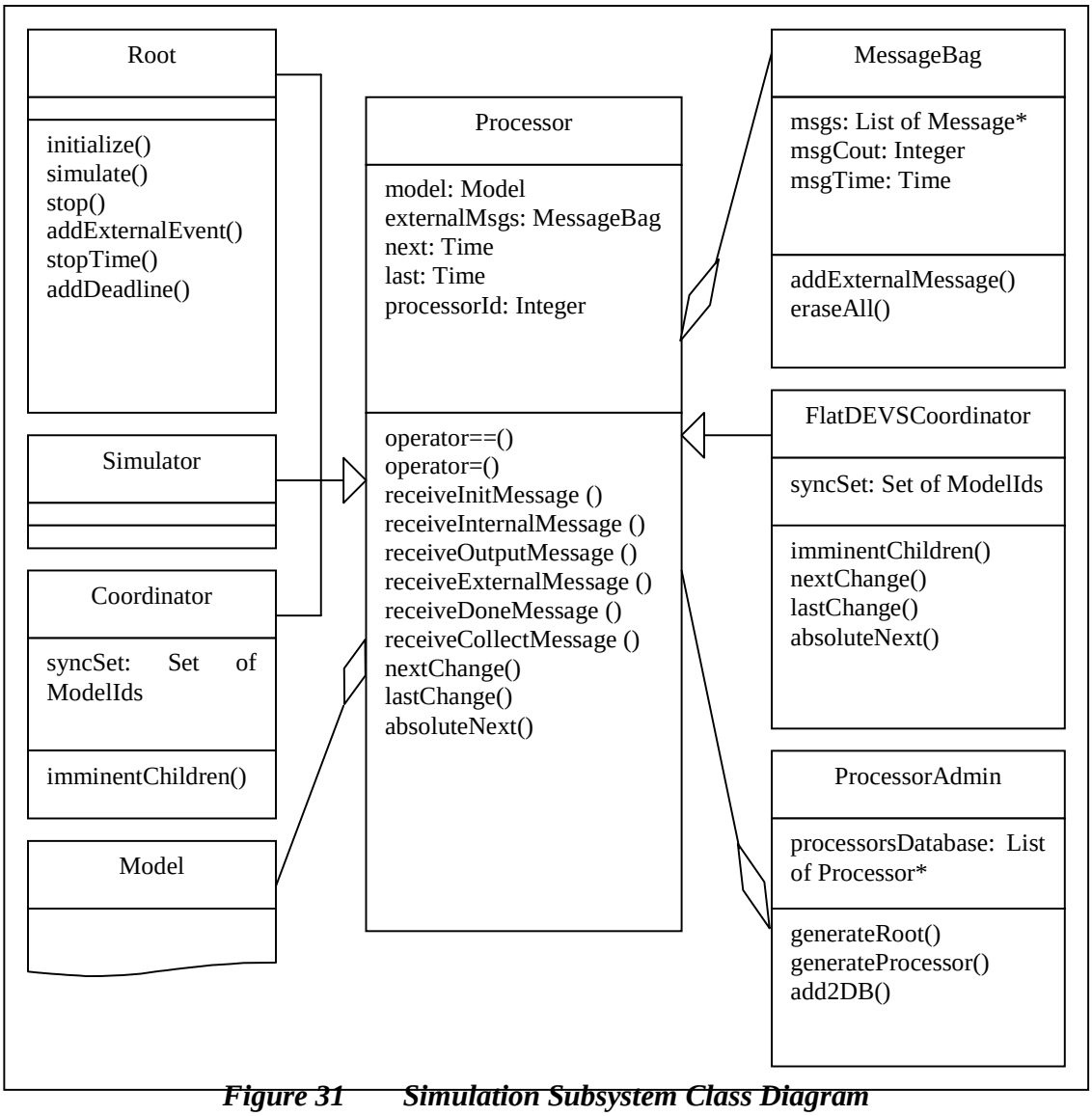

Figure 31 Simulation Subsystem Class Diagram

Furthermore, the Processor class defines the prototypes of handlers that respond to various DEVS messages, including initialization, internal and external state transition, output, collect and done messages. These methods in the `Process` class, however, are pure virtual functions which will be overloaded by the `Simulator` and `Coordinator` class.

The `Processor` class has four derived classes: `Simulator`, `Coordinator`, `FlatDEVSCoordinator`, and `Root`.

- The `Simulator` class implements the P-DEVS Atomic models, and, more precisely, it is responsible for invoking the atomic model's $\lambda(s)$, δ_{ext}, δ_{int}, δ_{con} functions. The message handler functions defined in the `Processor` class are virtual functions that are overloaded by the `Simulator` class. The algorithms of these message handlers have been described in section 3.4. Other than inheriting the data members and the methods from the `Processor` class, the Simulator class does not introduce any new ones.
- Similarly, the `Coordinator` class implements the P-DEVS Coupled models. `Coordinator` objects are responsible to carry out the simulation of the coupled models. The `Coordinator` class also overloads the message handlers defined in the `Processor` class. (The algorithms of these message handlers have been described in section 3.4.) Other than inheriting the data members and the methods from the `Processor` class, the `Coordinator` class also adds a new data member called `syncSet`, which serves as the synchronization set in P-DEVS. In addition, the new method `imminentChildren()` calculates the coupled models imminent children and updates the synchronization set.
- The Flattened Coordinator is implemented by the `FlatDEVSCoordinator` class. Its implementation is mainly the same as that of the `Coordinator` class, except that it overloads the `lastChange()`, `nextChange()` and `absoluteNext()` methods. Also, the Flattened Coordinator receives and sends messages directly from and to the Root Coordinator.
- The last subclass derived from `Processor` is `Root`. It represents the *Root Coordinator*. Its `simulate()` method starts the simulation by sending the very first initialization message to the *Top Coordinator*, and this method is invoked by the Main Simulator. Similarly, the `stop()` method is also used by the Main Simulator to stop or abort the entire simulation. The Root Coordinator is also responsible for interacting with the environment. Its `addExternalEvent()` method receives the incoming external events, either by reading from the External Events Table or by receiving it directly from the real world via real hardware ports in real-time mode. It then sends the corresponding External Messages to the

Top Coordinator. In addition, Root also advances the Global Simulation Time during every simulation cycle. In real-time simulation, it binds the simulation time with wall-clock time. In addition, its `addDealine()` method associates deadlines to external events, so that deadline validation can be performed for real-time DEVS.

The *Processors Manager*, which is implemented by the `ProcessorAdmin` class, manages the Processor class objects. Every Processor object is identified by its `Processor ID` data member. The `ProcessorAdmin` object is created by the Root Coordinator. It maintains the `Processors Database`, which is a hashing table of pointers to Simulator and Coupled class objects, so that actions, such as searching, can be performed upon those objects. The method `generateRoot()` is called in Root class constructor to create the Root Simulator. And the method `generateProcessor(Atomic or Coupled)`is called by the Models Manager during the Models Hierarchy Tree construction time to create Simulators and Coordinators, and it then calls the `add2DB()`method to add simulators and coordinators to the Processors Database.

4.6 Messaging Subsystem

The *Messaging Subsystem* is responsible for message delivery. Messages are used by simulators and coordinators to exchange data and synchronize activities. The nature and usage of messages are explained in section 3.4 where P-DEVS is discussed in detail.

The Messaging Subsystem consists of the Message Manager and various types of Message objects. Figure 32 shows the class diagram of the subsystem. Simulators and coordinators send messages via the *Messages Manager*, which is implemented by the `MessageAdmin` class. The `MessageAdmin` object is responsible for delivering messages among components (including both atomic and coupled). It is created by the Root Coordinator when it sends the very first initialization message to the Top

component. During its operation, it first buffers the incoming messages to its `Unprocessed Message Queue`, which is a queue of pointers to Message objects.

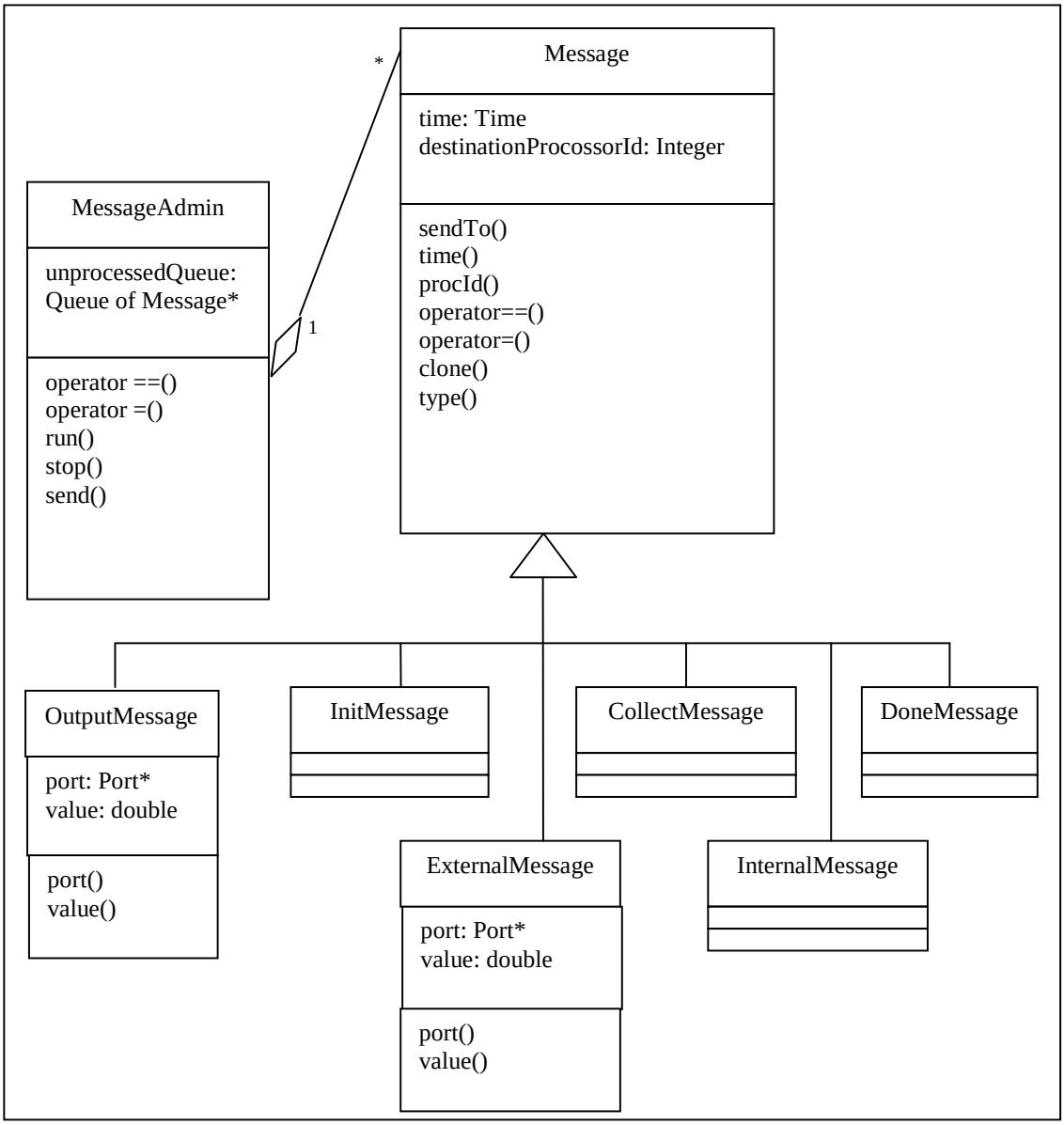

Figure 32 Messaging Subsystem Class Diagram

These messages are then processed by the Messages Manager in the FIFO (first-in-first-out) order. The `MessageAdmin` class provides the following public methods.

- *Send (message, modelID).* This method is used by the simulators and coordinators to send a *message* to the component specified by the *modelID*. The method simply adds the *message* to the FIFO Message Queue.
- *Run()*. This method is called by the Root Coordinator's `simulate()` method at the beginning of the simulation cycle. It loops through the `Unprocessed Message Queue` and sends out the messages by calling their `sendTo()` method defined in the Message class. The `Run()` method continuously checks the message queue, and it stops only when the `Stop()` method is called.
- *Stop()*. This method is used by the Main Simulator to stop the Message Manager when the simulation stops.

Messages are represented by the `Message` class. Its data member `time` records the creation time of the message, and it can be retrieved by the `time()` method. It also contains message receiver's ID (data member `destinationProcessorId`), which is used by its `sendTo()` method to deliver the message to the destination. This is achieved by invoking the receiving processor's message handler. A time-stamp (data member `time`) for the message and an associated *value* are also included in the `Message` object. The Message class has seven subclasses, each of which represents a particular message type. Table 5 lists these message types and their corresponding class names.

Message Type	**Message Symbol**	**Corresponding Class Name**
Initialization message	**I**	InitMessage
Collect message	**@**	CollectMessage
Internal message	*	InernalMessage
Done message	**D**	DoneMessage
External message	**q**	ExternalMessage
Output message	**y**	OutputMessage

Table 5 Various Types of Messages Supported by the Messaging Subsystem

The `type()` method in the Message class is a virtual function which is implemented by each of the subclasses to return the correct message type. While all other message types

contain only the time-stamp, message type, and destination ID, the output and external message also need to specify the message port and message value. Accordingly, the `OutputMessage` and `ExternalMessage` class contain two new data members: `port` and `value`. The receiving components of these two types of messages can retrieve the ports and message values by calling the port() and value() method respectively.

Chapter 5 Case Study

We have used E-CD++ to build a number of examples. In this chapter, we use one of them as a case study to demonstrate how to use E-CD++ to develop a complete embedded application. The example shows the creation of an Automated Manufacturing System (AMS), which evolves from a fully simulated version to a model with hardware-in-the-loop, and to a complete embedded application. To illustrate how E-CD++ is used to develop the AMS, we focus on the discussion of one particular development phase where the AMS is a hybrid system in which the simulated components are mixed and interact with hardware surrogates. We will demonstrate how the AMS is modelled as a hybrid system and how GGAD notation is used for the modelling. We will then show how the model is executed by E-CD++ in an embedded environment. We will examine deadline checking, flattened coordinator performance, and confluent function execution.

5.1 Modelling the AMS

Figure 33 shows the physical layout of the AMS, which consists of four workstations and two conveyor belts to transport the products (A and B). Each of the four workstations performs a specific task on a given product. The product is partially built when it goes through each of the workstations. The AMS also uses two conveyor belts moving in opposite directions carrying the products to the scheduled workstation. The production cycle is organized by a scheduler, which depends on the type of piece being assembled. The scheduler determines which station should receive and work on the product. The AMS has real-time constraints (i.e., the product must be delivered to and departure from the predetermined workstations at the exact scheduled time).

The AMS in this case study consists of two conveyors (each conveyor has an engine and a sensor controller), one controller unit, one scheduler, one display controller, and two notification bells, as shown in Figure 34.

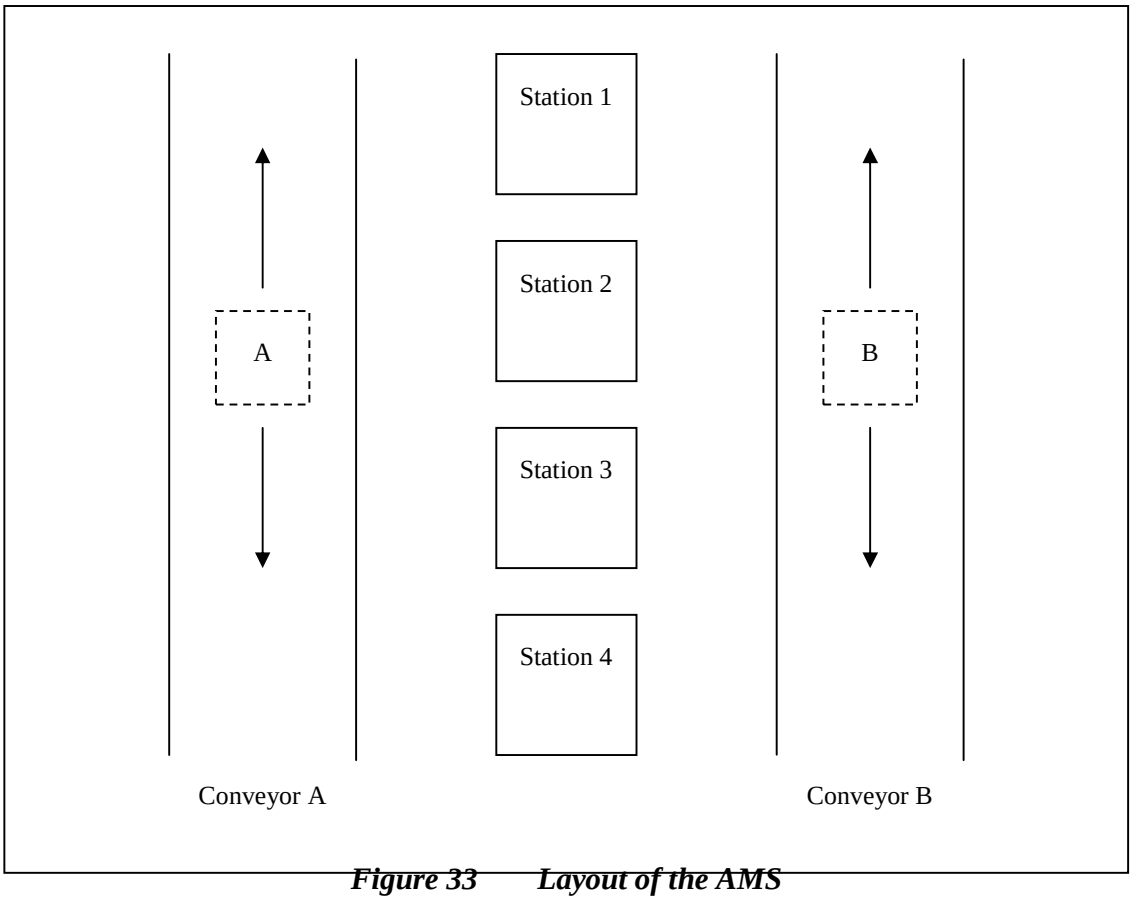

Figure 33 Layout of the AMS

The behaviour of each AMS component is described as follows:

- The *Scheduler* contains the working schedule as for which stations have to work on a specific product. It sends the schedule to the Control Unit.

- The *Control Unit (CU)* is the most complex part in AMS. It receives the schedule from the Scheduler and controls the two conveyors. Figure 35 represents a block diagram of the CU.

- The Scheduler sends schedules (external events) to ports *station_ij*, indicating that the product in conveyor belt *j* has to be sent to station *i*. Events received via *sensor_ij* indicate that the product in conveyor *j* has reached station *i*. Consequently, the CU activates or deactivates the engine of the corresponding

conveyor (via *direction_j* and *activate_j*). It can also signal the Display Controller when the conveyor belt starts moving or a product reaches a new station (via *direction_display_j* and *station_display_j*).

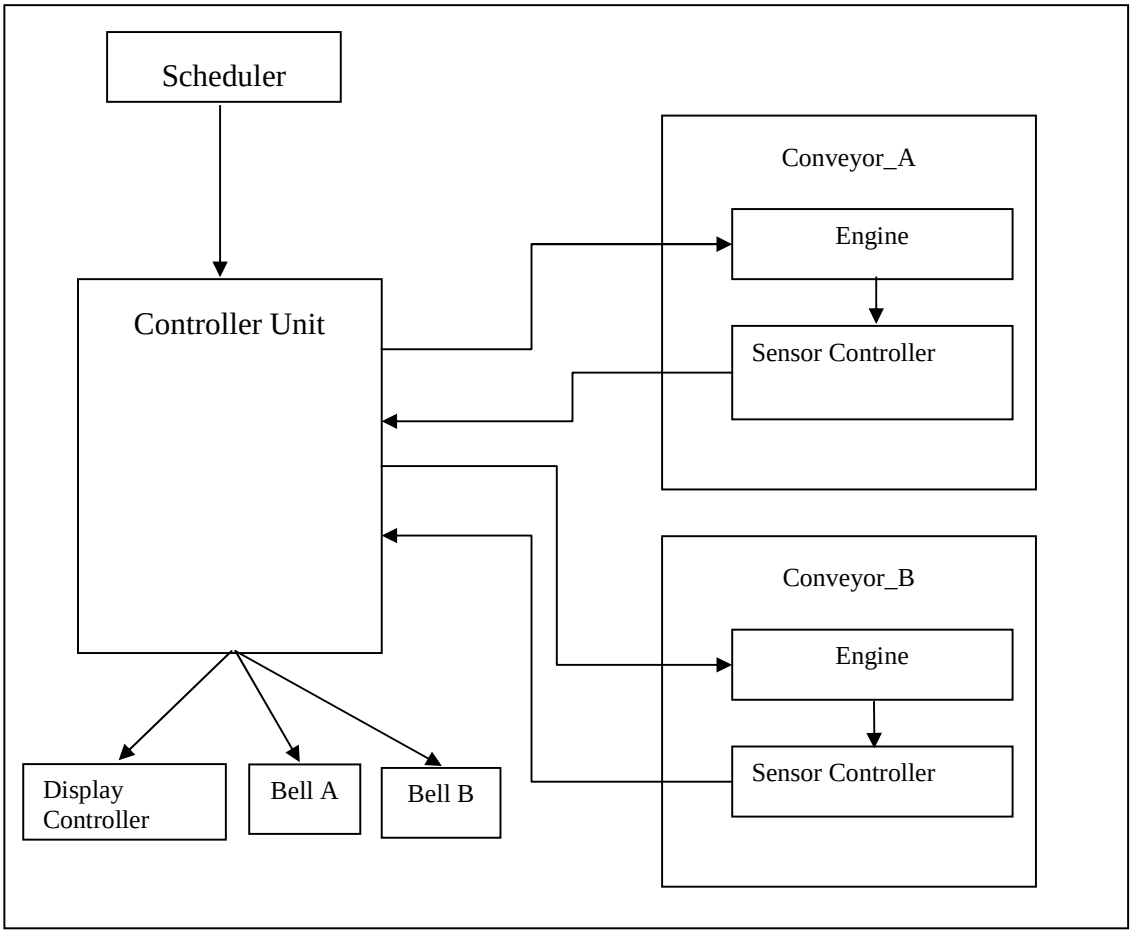

Figure 34 Scheme of the AMS

- The *Conveyor* contains a Sensor Controller and an Engine. The *Engine* drives the conveyor belt. It can move in both directions, and its movements are controlled by CU. The *Sensor Controller* receives the working piece's displacement location from the engine, and forwards this information to CU, which then determines the next action for the engine (e.g., deactivation if the piece has reached the destination station, or activation if otherwise).

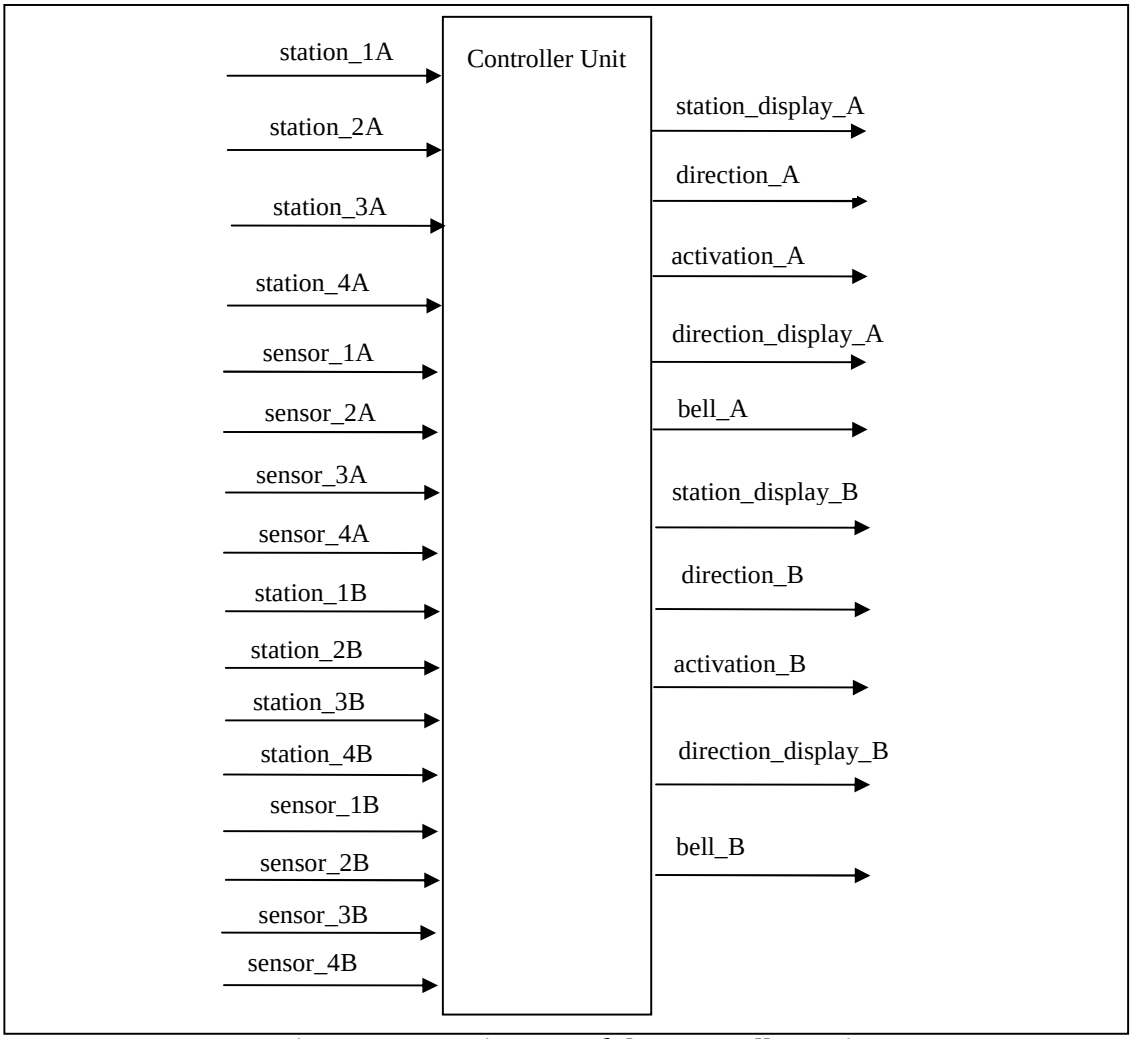

Figure 35 Diagram of the Controller Unit

- The *Display Controller* handles the digital display (showing the location of the piece in each conveyor belt), based on the signals from the Controller Unit. It displays the moving directions of the 2 conveyers and the position statuses of the moving products. The moving directions are displayed as the output value 0, 1, or 2 indicating stopping, moving forward, or moving backward, respectively. The position status of a moving piece is also shown as value *ij*, indicating that the product in conveyor *j* has reached station *i*. The Display Controller also has four LEDs output ports, namely *Led1*, *Led2*, *Led3*, and *Led4*. These LEDs are destination indicators, and each LED port is associated with one station. If, for instance, Led3 is on (with value being 1), that means a product needs to be

transported to station 3. And the LED will be turned off (with value being 0) when the product reaches its destination.

- There are two *Notification Bells*, one for each conveyor. Once the conveyor finishes transporting the product to the destination station, the bell associated with that conveyor will ring indicating the completion. (The actual completion time will then be checked against the specified deadline defined by the Scheduler.)

5.1.1 Hybrid System Modelling

After defining a fully simulated version of the model, we developed the AMS using hardware-in-the-loop. To best demonstrate the involvement of E-CD++ in this development, this case study chooses to study an intermediate development phase where the AMS is a hybrid system in which the simulated components are mixed and interact with the real hardware parts. In that development phase, the real hardware parts are: the Scheduler, the Display Controller, and the 2 Notification Bells. And the rest of the AMS components are still in simulation mode. Table 6 summarizes the model composition of the hybrid system. (Note that we model the Sensor Controller using graphical notations. We will explain how this is done in the next section.)

Component Name	Component Type	DEVS Model Name	Graphical Notation Used	Component Quantity
Scheduler	Real hardware	N/A	N/A	1
Display Controller	Real hardware	N/A	N/A	1
Notification Bell	Real hardware	N/A	N/A	1
Controller Unit	Atomic model	CU	No	1
Conveyor	Coupled model	ConveyorA, ConveyorB	No	2
Engine	Atomic model	EngA, EngB	No	2
Sensor Controller	Atomic model	ScA, ScB	Yes	2

Table 6 The Hybrid AMS Model

The resulting hardware-in-the-loop configuration of the hybrid system is shown in Figure 36. The Scheduler, Display Controller and Notification Bells interact with the simulated

Controller Unit through the real I/O ports on the development board (i.e., SBC). The Controller Unit interacts with the four hardware components the same way as if they were simulated atomic components.

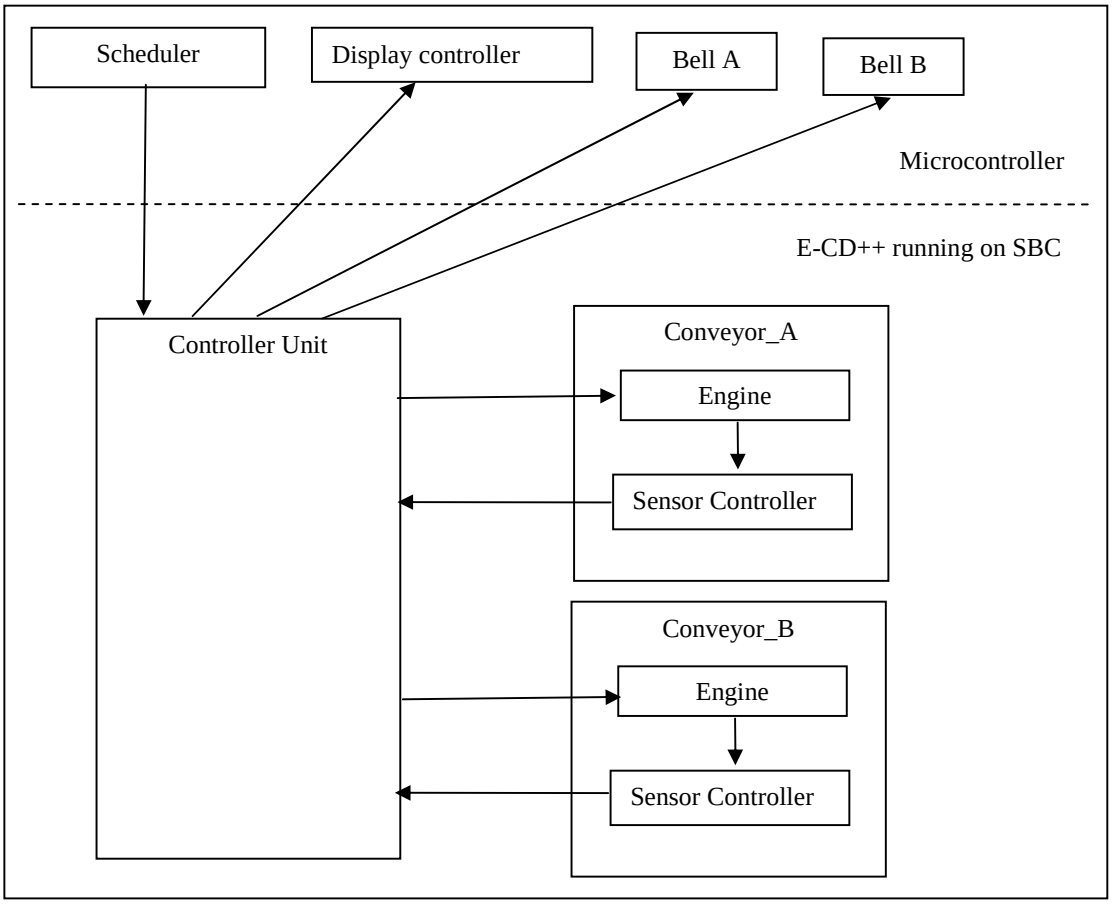

Figure 36 Hybrid AMS Scheme (scheduler, display and bells in hardware)

In order to model this hybrid system, we need to define the DEVS model for the simulated portion of AMS and identify the model's I/O ports that interface with real hardware. We first need to define the component hierarchy and the port linkage among the components. This is defined in the DEVS model file, shown in Figure 37. The DEVS model file defines CU as an atomic model (line 57 - 58) and Conveyor as a coupled model (line 35 – 56), which consists of two atomic models: Engine (line 59 - 62) and Sensor (line 63 - 66). Note that the two Sensor Controllers are defined in GGAD graphical notation, which are stored in sensorA.cdd (line 64) and sensorB.cdd (line 66)

file. (We will use the GGAD model to examine E-CD++ graphical modelling capability. More details will follow in the next section.)

```
1.  [top]
2. components : conveyorA conveyorB cu@ECU
3. in : btn1A btn2A btn3A btn4A btn1B btn2B btn3B btn4B
4. out : led1 led2 led3 led4 stn_disp_A stn_disp_B dirn_disp_A dirn_disp_B
   bellA bellB

5. Link : btn1A b1A@cu
6. Link : btn2A b2A@cu
7. Link : btn3A b3A@cu
8. Link : btn4A b4A@cu
9. Link : btn1B b1B@cu
10. Link : btn2B b2B@cu
11. Link : btn3B b3B@cu
12. Link : btn4B b4B@cu

13. Link : activate_A@cuactivate_A@conveyorA
14. Link : direction_eng_A@cudirection_eng_A@conveyorA
15. Link : activate_B@cuactivate_B@conveyorB
16. Link : direction_eng_B@cudirection_eng_B@conveyorB

17. Link : s1A@conveyorA s1A@cu
18. Link : s2A@conveyorA s2A@cu
19. Link : s3A@conveyorA s3A@cu
20. Link : s4A@conveyorA s4A@cu
21. Link : s1B@conveyorB s1B@cu
22. Link : s2B@conveyorB s2B@cu
23. Link : s3B@conveyorB s3B@cu
24. Link : s4B@conveyorB s4B@cu

25. Link : l1@cu led1
26. Link : l2@cu led2
27. Link : l3@cu led3
28. Link : l4@cu led4
29. Link : ringBellA@cu bellA
30. Link : ringBellB@cu bellB

31. Link : station_display_A@cu stn_disp_A
32. Link : station_display_B@cu stn_disp_B
33. Link : direction_display_A@cu dirn_disp_A
34. Link : direction_display_B@cu dirn_disp_B

35. [conveyorA]
36. components : engA@engine scA@sensorboxA
37. in : activate_A direction_eng_A
38. out : s1A s2A s3A s4A
39. Link : activate_Astartstop@engA
40. Link : direction_eng_Aengdirection@engA
41. Link : sen1A@scA s1A
42. Link : sen2A@scA s2A
43. Link : sen3A@scA s3A
44. Link : sen4A@scA s4A
45. Link : floor@engA s1A_eng@scA

46. [conveyorB]
47. components : engB@engine scB@sensorboxB
48. in : activate_B direction_eng_B
49. out : s1B s2B s3B s4B
50. Link : activate_Bstartstop@engB
```

```
51. Link : direction_eng_Bengdirection@engB
52. Link : sen1B@scB s1B
53. Link : sen2B@scB s2B
54. Link : sen3B@scB s3B
55. Link : sen4B@scB s4B
56. Link : floor@engB s1B_eng@scB

57. [cu]
58. preparation : 00:00:00:200

59. [engA]
60. preparation : 00:00:01:000

61. [engB]
62. preparation : 00:00:01:000

63. [scA]
64. source : sensorA.cdd

65. [scB]
66. source : sensorB.cdd
```

Figure 37 Definition of the AMS system in E-CD++

The DEVS model file in Figure 37 also defines the input ports (line 3) that connect to the Scheduler and the output ports (line 4) that connect to the Display Controller and the Notification Bells. Through these I/O ports, the simulated models interact directly with the hardware components.

Figure 38 is the graphical presentation of the AMS model file, which shows the port linkage more intuitively. From Figure 38, we see that the Scheduler hardware sends the command to CU by writing to one of its 8 input ports, indicating the destination station. The CU then forwards the command to one of the Conveyors via the “activate” and “direction” port links shown in the figure. The Conveyor then forwards the command to the Engine through the ports connecting these two components. While executing the command, the Engine outputs its operation status to the Sensor via its output port. The Sensor then forwards the Engine status to the Conveyor, which once again forwards the message to the CU, which then sends the transportation status to the Display Controller through its output ports. If the working piece has reached its destination, the CU will also notify the Notification Bells. All these information exchanges are done via the port links shown in Figure 38.

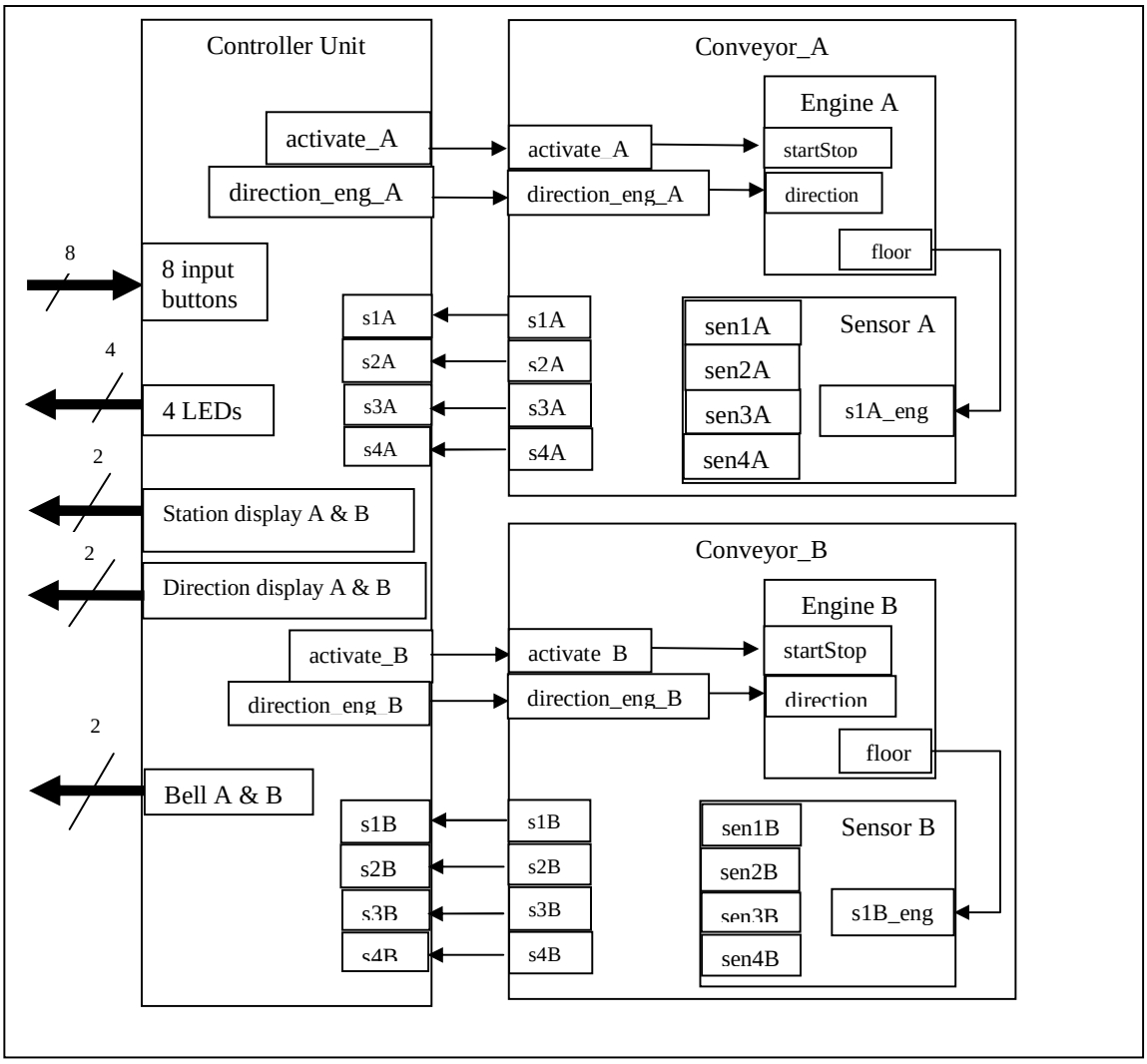

Figure 38 Modelling Scheme of the Simulated Part of AMS

5.1.2 GGAD Graphical Modelling

In this experiment, we define the Sensor Controller in GGAD graphical notations. To illustrate, Figure 39 shows the graphical notation of Sensor Controller A, or `scA`. `scA` has one input port (`s1A`) and four output ports (sen1A, sen2A, sen3A and sen4A). The input port connects to the conveyor's engine. When the conveyor delivers the working product to a particular workstation, the engine will send the workstation number to `scA` through its input port `s1A`. The 4 output ports of scA correspond to the 4 workstations respectively. After receiving the input from the engine, `scA` will toggle the output port which corresponds to the input workstation number.

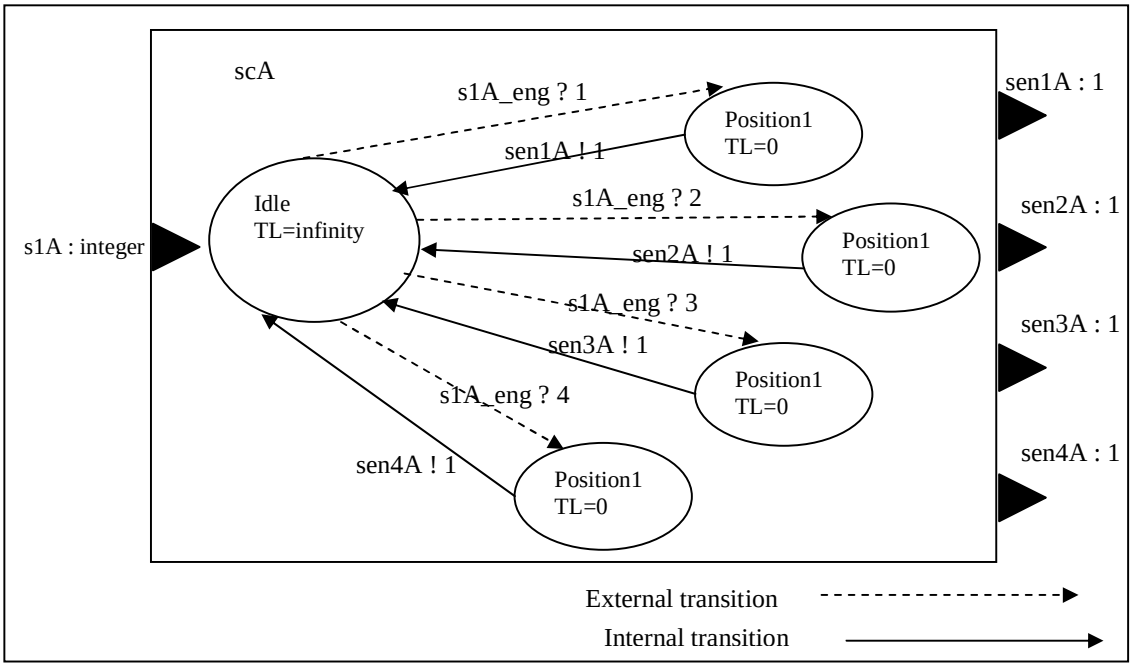

Figure 39 GGAD Graphical Notation of the Sensor Controller

The graphical notation will be automatically converted to the GGAD model file. Figure 40 shows the generated GGAD model file of Sensor Controller A.

```
1.   [scA]
2.   in: s1A_eng
3.   out: sen1A sen2A sen3A sen4A
4.   var : cur_value last_value
5.   state: idle position1 position2 position3 position4
6.   initial: idle
7.   ext: idle position1 equal(s1A_eng, 1)?1 {cur_value = s1A_eng;}
8.   ext: idle position2 equal(s1A_eng, 2)?1 {cur_value = s1A_eng;}
9.   ext: idle position3 equal(s1A_eng, 3)?1 {cur_value = s1A_eng;}
10.  ext: idle position4 equal(s1A_eng, 4)?1 {cur_value = s1A_eng;}
11.  int: position1 idle sen1A!1 {last_value = cur_value;}
12.  int: position2 idle sen2A!1 {last_value = cur_value;}
13.  int: position3 idle sen3A!1 {last_value = cur_value;}
14.  int: position4 idle sen4A!1 {last_value = cur_value;}
15.  idle: infinite
16.  position1: 0:0:0:0
17.  position2: 0:0:0:0
18.  position3: 0:0:0:0
19.  position4: 0:0:0:0
20.  cur_value: 1
21.  last_value: 1
```

Figure 40 GGAD Model File of the Sensor Controller

Sensor Controller A has connections to Engine A and CU, as shown in Figure 38. We can see, in Figure 38, that `scA` has one input port (`s1A_eng`) and 4 output ports (`sen1A`,

sen2A, sen3A, and sen4A). These ports are also defined in the GGAD model file (line 2–3). The scA's external transition function is defined in line 7 – 10 of Figure 40. When an external message arrives at the input port s1A_eng, the input value N is an integer in range [1, 4], denoting the station where the product currently arrives. The external transition function checks this message value, using GGAD built-in function equal (s1A_eng, N)?1, and moves the model state from idle to position{N}.

Since the elapsed time of the position{N} state is zero (line 16–19), the internal transition function (line 11 – 14) is triggered immediately after the external transition. It sends the output value of 1 to output port sen{N}A, where N is the same as that in position{N}, and moves the model state back to idle, which is the passive state (as defined in line 15).

E-CD++ is able to load the Sensor Controller GGAD model file and simulate the model behaviour described above. This graphical modelling capability reduces our modelling efforts. Without this technique, the modeller has to write a C++ class for the Sensor Controller. For comparison, a C++ implementation is shown in Figure 41.

```
SensorBoxA::SensorBoxA( const std::string &name ) : Atomic( name ),
s1A_eng( addInputPort( "s1A_eng" )), sen1A( addOutputPort( "sen1A" ) ),
sen2A( addOutputPort( "sen2A" )), sen3A( addOutputPort( "sen3A" )),
sen4A( addOutputPort( "sen4A")), preparationTime( 0,0,0,0 ) {
  if( time != "" )    preparationTime = time ;
}

Model &SensorBoxA::initFunction() {
        cur_value = last_value = 1;
        return *this ;
}

Model &SensorBoxA::externalFunction( const ExternalMessage &msg ) {
    // New value arrived on the input port
    if( msg.port() == s1A_eng )     {
        cur_value = msg.value();
        holdIn( Atomic::active, preparationTime );
    }
        return *this;
}

Model &SensorBoxA::internalFunction( const InternalMessage & ) {
    passivate();
    return *this ;
}
```

```
Model &SensorBoxA::outputFunction( const InternalMessage &msg ) {
  if (last_value != cur_value)   {
     if (cur_value == 1){  sendOutput( msg.time(), sen1A , 1 ); }
     else if (cur_value == 2){  sendOutput( msg.time(), sen2A , 1 ); }
     else if (cur_value == 3){  sendOutput( msg.time(), sen3A , 1 ); }
     else if (cur_value == 4){  sendOutput( msg.time(), sen4A , 1 ); }
     else last_value = cur_value;
  }
  else
    return *this ;
}
```

Figure 41 The Sensor Class

We can see that the GGAD notation is also simpler than the C++ code. Furthermore, the GGAD model file can be formally validated, whereas the C++ code cannot.

5.2 Model Execution using E-CD++

This section defines an experimental frame for the AMS simulation. We use it to test the hybrid AMS. As explained in the previous section, the hybrid AMS has four hardware components – the Scheduler, the Display Controller and the two notification bells, and the rest of the components are in simulation. Our experiment runs tests on every component.

The experiment starts with creating the work item schedule. The schedule is generated by the Scheduler. It defines which stations have to work on a specific product at what time. The Controller Unit (CU) controls the movement of the Conveyors according to the schedule. So, the schedule serves the same role as an external event file sent to the CU. Figure 42 is the schedule we use for this experiment.

Start time	*Associated deadline*	*input port*	*associated output port*	*value*
00:00:02:100	00:00:05:300	Btn3A	bellA	1
00:00:06:130	00:00:10:300	Btn4B	bellB	1

Figure 42 An experimental event file generated by the scheduler

The initial conditions of the experiment are: (1) the product is always placed on the first workstation of each conveyor belt, and (2) the experiment starts at time 00:00:00:000

(The time format is hh:mm:ss:msec). The values in the schedule are relative to the initial conditions. In Figure 42, there are two scheduled tasks:

- The first task is scheduled to start at time 00:00:02:100. It requires Conveyor A to move the product from workstation 1A to workstation 3A before the deadline 00:00:05:300. This is done by the Scheduler sending a signal to the CU's input port `btn3A` (which corresponds to the workstation 3A) at time 00:00:02:100. The CU's output port `bellA` is used for deadline checking. When the task competes, the CU rings Bell A (by writing value 1 to its output port `bellA`). E-CD++ compares this completion time against the specified deadline.

- Similarly, the second task is scheduled to start at time 00:00:06:130. It requires conveyor B to move its product from workstation 1B to workstation 4B before the deadline 00:00:10:300.

The CU component running on the SBC interacts with the Scheduler chip via I/O ports and sends the simulation results to the Display Controller. Figure 43 shows the experiment results displayed by the Display Controller.

actual output time (physical time)	*Associated deadline*	*result*	*output port*	*value*
00:00:02:300		**No deadline**	Led3	1
00:00:02:300		**No deadline**	dirn_disp_a	1
00:00:03:350		**No deadline**	stn_disp_a	21
00:00:04:350		**No deadline**	stn_disp_a	31
00:00:04:350		**No deadline**	dirn_disp_a	0
00:00:04:350		**No deadline**	Led3	0
00:00:04:360	00:00:05:300	**Succeeded**	Bell_A	1
00:00:06:330		**No deadline**	Led4	1
00:00:06:330		**No deadline**	dirn_disp_b	1
00:00:07:380		**No deadline**	stn_disp_b	22
00:00:08:380		**No deadline**	stn_disp_b	32
00:00:09:380		**No deadline**	Stn_disp_b	42
00:00:09:380		**No deadline**	dirn_disp_b	0
00:00:09:380		**No deadline**	Led4	0
00:00:09:380	00:00:100:300	**Succeeded**	Bell_B	1

Figure 43 Simulation results displayed by the Display Controller

The result in the first column of Figure 43 shows the *actual time* at which the output has been sent, which is the wall-clock value at that time (the time elapsed since the beginning of the simulation execution). The second column shows the *associated deadline time* for

the given event. The third column indicates whether the deadline has been met (*i.e.* the actual output time ≤ the associated deadline). Finally, the *output ports* and their *output values* are shown in the remaining two columns, respectively.

5.2.1 Executions of Simulated Components

As mentioned earlier, the hybrid system has three atomic models running in simulation mode: `Engine`, `Sensor`, and `Controller_Unit`. E-CD++ provides a runtime environment for these 3 models to interact each other. The `Engine` and `Sensor` model work together to constitute the behaviour of the coupled model `Conveyor`. The `Engine` model is written in C++. Its inputs ports are connected to the `Controller Unit`. When an external message is sent from the CU, the Engine's external transition function (Figure 44) will be executed. The external transition function mainly sets the `Engine` model to new states based on the input values.

```
Model &Engine::externalFunction( const ExternalMessage &msg ) {
  if ( msg.port() == startstop )  {
    if ( (msg.value() == 1) && ( !working ) ) {
      if (cur_direction == 1)      {
         ready2Up = true;
         holdIn( Atomic::active, preparationTime2Start );
      }
      else if (cur_direction == 2) {
         ready2Down = true;
         holdIn( Atomic::active, preparationTime2Start );
      }
    }
    else if ( (msg.value() == 0) && ( working ) ) {
      ready2Stop = true;
      holdIn( Atomic::active, preparationTime2Stop );
    }
  }
  // Second, is it a direction request?
  else if (msg.port() == engdirection) {
    if (!working)           {
      cur_direction = msg.value();
    }
  }
  return *this;
} // End of dExt
```

Figure 44 External Transition Function of the Engine Model

After the external transition is finished, E-CD++ will execute the `Engine's` Internal Transition Function (Figure 45), which will set the new *ta*(s).

```
Model &Engine::internalFunction( const InternalMessage & ) {
  // Was it ready to stop? -> STOP
  if ( ready2Stop )   {
    working = 0;
    cur_direction = 0;
    ready2Stop = false;
  }
  // Was it ready to go forward? -> GO FORWARD
  else if ( ready2Up )   {
    working = 1;
    cur_direction = 1;
    ready2Up = false;
    next_floor = cur_floor + 1;
    holdIn( Atomic::active, floorTime );
  }
  // Was it ready to back? -> GO BACKWARD
  else if ( ready2Down )   {
    working = 1;
    cur_direction = 2;
    ready2Down = false;
    next_floor = cur_floor - 1;
    holdIn( Atomic::active, floorTime );
  }
  // This is a new station now!   Going forward?
  else if (working && (cur_direction==1))   {
    cur_floor = next_floor;
    next_floor = cur_floor + 1;
    holdIn( Atomic::active, floorTime );
  }
  // Going backwards?
  else if (working && (cur_direction==2)) {
     cur_floor = next_floor;
     next_floor = cur_floor - 1;
     // Next transition depends on time that takes to go back 1 station,
     // unless external event received
     holdIn( Atomic::active, floorTime );
  }
  else
     passivate();
  return *this ;
} // End of dInt
```

Figure 45 Internal Transition Function of the Engine Model

When *ta(s)* is elapsed, E-CD++ will execute `Engine's` Output Function (Figure 46), which sends an external message to the `Sensor` model (which is built in GGAD). Note that, via this external message, the `Engine` model's activities are synchronized with the `Sensor` model. This is how these two models work together.

```
Model &Engine::outputFunction( const InternalMessage &msg ) {
  // If this is not happening while ready to stop,
  // it is a station forward or backward, then issue the value
```

```
  if (!ready2Stop)   {
    // Working and going forward, inform new station
    if ( working && (cur_direction==1) )     {
      // Send the next station, that will be set
      // as current station in dInt, immediately
      sendOutput( msg.time(), floor, next_floor ) ;
    }
    // Working and going backwards, inform new station
    else if ( working && (cur_direction==2) )     {
      // Send the next station, that will be set as
      // current station in dInt, immediately
      sendOutput( msg.time(), floor, next_floor ) ;
    }
  }
  return *this;
}
```

Figure 46 Output Function of the Engine Model

When conducting this experiment, we recorded the messages generated during E-CD++ runtime. The message log is an important device to trace and examine the internal activities of the simulated models, as well as their interactions. It serves as a supplement to the output file (shown in Figure 43) for verification purposes. To illustrate how the message log can be used for verification, consider the sample portion of the message log shown in Figure 47.

- Line 1 shows that the simulation started at time 00:00:00:000, which is what we expected. (Time is wall-clock time.)
- Line 3 shows that an external event arrived to port `btn3a` at time 00:00:02:100. This was the first scheduled event by generated the Scheduler.
- Line 5 shows that the `Controller_Unit`'s external transition function is executed at the same time to handle this external event.
- Line 13 and 14 are where the `Engine's` external transition function is called.
- Line 33 is where the `Engine's` output function sends an external message to the `Sensor`.

```
1.   MSG: I / 00:00:00:000 / Root(00) TO flattop(01)
2.   MSG: D / 00:00:00:000 / flattop(01) / ... TO Root(00)
3.   MSG: X / 00:00:02:100 / Root(00) / btn3a /      1.00000 TO flattop(01)
4.   MSG: * / 00:00:02:100 / Root(00) TO flattop(01)
5.   MSG: X / 00:00:02:100 / flattop(01) / b3a /      1.00000 TO cu(08)
6.   MSG: * / 00:00:02:100 / flattop(01) TO cu(08)
7.   MSG: D / 00:00:02:100 / cu(08) / 00:00:00:200 TO flattop(01)
8.   MSG: D / 00:00:02:100 / flattop(01) / 00:00:00:200 TO Root(00)
9.   MSG: @ / 00:00:02:300 / Root(00) TO flattop(01)
10.  MSG: @ / 00:00:02:300 / flattop(01) TO cu(08)
11.  MSG: Y / 00:00:02:300 / flattop(01) / led3 /      1.00000 TO Root(00)
12.  MSG: Y / 00:00:02:300 / flattop(01) / dirn_disp_a / 1.00000 TO Root(00)
13.  MSG: X / 00:00:02:300 / flattop(01) / engdirection/1.00000 TO enga(03)
14.  MSG: X / 00:00:02:300 / flattop(01) / startstop / 1.00000 TO enga(03)
15.  MSG: D / 00:00:02:300 / cu(08) / ... TO flattop(01)
16.  MSG: D / 00:00:02:300 / flattop(01) / 00:00:00:000 TO Root(00)
17.  MSG: * / 00:00:02:300 / Root(00) TO flattop(01)
18.  MSG: * / 00:00:02:300 / flattop(01) TO enga(03)
19.  MSG: * / 00:00:02:300 / flattop(01) TO cu(08)
20.  MSG: D / 00:00:02:300 / enga(03) / 00:00:00:050 TO flattop(01)
21.  MSG: D / 00:00:02:300 / cu(08) / ... TO flattop(01)
22.  MSG: D / 00:00:02:300 / flattop(01) / 00:00:00:050 TO Root(00)
23.  MSG: @ / 00:00:02:350 / Root(00) TO flattop(01)
24.  MSG: @ / 00:00:02:350 / flattop(01) TO enga(03)
25.  MSG: D / 00:00:02:350 / enga(03) / ... TO flattop(01)
26.  MSG: D / 00:00:02:350 / flattop(01) / 00:00:00:000 TO Root(00)
27.  MSG: * / 00:00:02:350 / Root(00) TO flattop(01)
28.  MSG: * / 00:00:02:350 / flattop(01) TO enga(03)
29.  MSG: D / 00:00:02:350 / enga(03) / 00:00:01:000 TO flattop(01)
30.  MSG: D / 00:00:02:350 / flattop(01) / 00:00:01:000 TO Root(00)
31.  MSG: @ / 00:00:03:350 / Root(00) TO flattop(01)
32.  MSG: @ / 00:00:03:350 / flattop(01) TO enga(03)
33.  MSG: X / 00:00:03:350 / flattop(01) / s1a_eng /      2.00000 TO sca(04)
34.  MSG: D / 00:00:03:350 / enga(03) / ... TO flattop(01)
35.  MSG: D / 00:00:03:350 / flattop(01) / 00:00:00:000 TO Root(00)
36.  MSG: * / 00:00:03:350 / Root(00) TO flattop(01)
37.  MSG: * / 00:00:03:350 / flattop(01) TO enga(03)
38.  MSG: * / 00:00:03:350 / flattop(01) TO sca(04)
39.  MSG: D / 00:00:03:350 / enga(03) / 00:00:01:000 TO flattop(01)
40.  MSG: D / 00:00:03:350 / sca(04) / 00:00:00:000 TO flattop(01)
41.  MSG: D / 00:00:03:350 / flattop(01) / 00:00:00:000 TO Root(00)
42.  ……
```

Figure 47 Sample Message Log Trace

By examining the messages, we can verify if the activities are done at the right time with the right values.

5.2.2 Measurements on Flattened Coordinator's Performance

We want to use this experiment to measure the performance improvements gained from the flatten coordinator technique. To do that, we first compare the flattened model

hierarchy with the original model hierarchy. The original AMS model, shown in Figure 38, is a two-level hierarchy. The Controller and the two Conveyors are at the upper level, and the Engines and Sensors are at the bottom level. Without Flattened Coordinator technique, messaging needs to go through this two-level model hierarchy. For instance, suppose that a user presses a button (an input port on the Controller Unit) that triggers the activation of Engine A. To simulate this event, the Controller Unit simulator sends an external message from its output port `activate_A` to the Conveyor_A models input port `activate_A`. When the coordinator conveyor_A receives this message, it then sends an external message from its input port `activate_A` to the Engine_A model's input port `startStop`, which triggers Engine_A to start. In this simulation example, two messages need to be generated before Engine A can be activated. We can see, from this example, that in order to complete the simulation, messages must be generated at every level of the model hierarchy. Therefore, the performance will be improved if we can eliminate middle levels in the hierarchy.

In comparison, the Flattened Coordinator technique flattens the AMS model hierarchy by eliminating the coordinators and hence reducing the number port links. The technique rewires any port link that link to a coupled model directly to the far-end atomic model.. For example, after the rewiring, the Controller Unit's port `activate_A` is directly linked to the Engine A's `startStop` port. Also, the two coupled models, conveyor A and B, are eliminated. Moreover, the technique also rewires any atomic model's output port that originally links to a coupled model directly to the far-end atomic model. The two Sensors output ports, for example, are directly linked to the Controller Unit, eliminating the intermediate links to the Conveyors' ports. From the comparison, we observe that the flattened model has less port links than the original model, which implies that the simulation will also generate less number of messages if we use the flattened model.

The simulator's performance is measured by the number of messages it generates during the simulation. The fewer the messages, the better the performance of the simulator.

Figure 48 shows a portion of the message log collected during the AMS simulation using the flattened coordinator technique.

```
MSG: I / 00:00:00:000 / Root(00) TO flattop(01)
MSG: D / 00:00:00:000 / flattop(01) / ... TO Root(00)
MSG: X / 00:00:02:100 / Root(00) / btn3a /      1.00000 TO flattop(01)
MSG: * / 00:00:02:100 / Root(00) TO flattop(01)
MSG: X / 00:00:02:100 / flattop(01) / b3a /      1.00000 TO cu(08)
MSG: * / 00:00:02:100 / flattop(01) TO cu(08)
MSG: D / 00:00:02:100 / cu(08) / 00:00:00:200 TO flattop(01)
MSG: D / 00:00:02:100 / flattop(01) / 00:00:00:200 TO Root(00)
MSG: @ / 00:00:02:300 / Root(00) TO flattop(01)
MSG: @ / 00:00:02:300 / flattop(01) TO cu(08)
MSG: Y / 00:00:02:300 / flattop(01) / led3 /      1.00000 TO Root(00)
MSG: Y / 00:00:02:300 / flattop(01) / dirn_disp_a /      1.00000 TO Root(00)
MSG: X / 00:00:02:300 / flattop(01) / engdirection /      1.00000 TO enga(03)
MSG: X / 00:00:02:300 / flattop(01) / startstop /      1.00000 TO enga(03)
MSG: D / 00:00:02:300 / cu(08) / ... TO flattop(01)
MSG: D / 00:00:02:300 / flattop(01) / 00:00:00:000 TO Root(00)
MSG: * / 00:00:02:300 / Root(00) TO flattop(01)
MSG: * / 00:00:02:300 / flattop(01) TO enga(03)
MSG: * / 00:00:02:300 / flattop(01) TO cu(08)
MSG: D / 00:00:02:300 / enga(03) / 00:00:00:050 TO flattop(01)
MSG: D / 00:00:02:300 / cu(08) / ... TO flattop(01)
MSG: D / 00:00:02:300 / flattop(01) / 00:00:00:050 TO Root(00)
MSG: @ / 00:00:02:350 / Root(00) TO flattop(01)
MSG: @ / 00:00:02:350 / flattop(01) TO enga(03)
……
```

Figure 48 Message Log Generated During the AMS Simulation

To measure the performance improvements made by the Flattened Coordinator technique, we used the AMS event file in Figure 42 in our experiment. We compared the number of messages generated during the simulation using the Flattened Coordinator technique with that generated by not using the technique. There are 257 messaged generated when the technique is used, compared with 385 messages generated when otherwise. So, the performance improvement ratio is 33%.

We now compare the performance improvement ratio obtained from the experimental results with the theoretical value. The original AMS component hierarchy contains 7 nodes, and this number is reduced to 5 by the Flattened Coordinator technique (Figure 49).

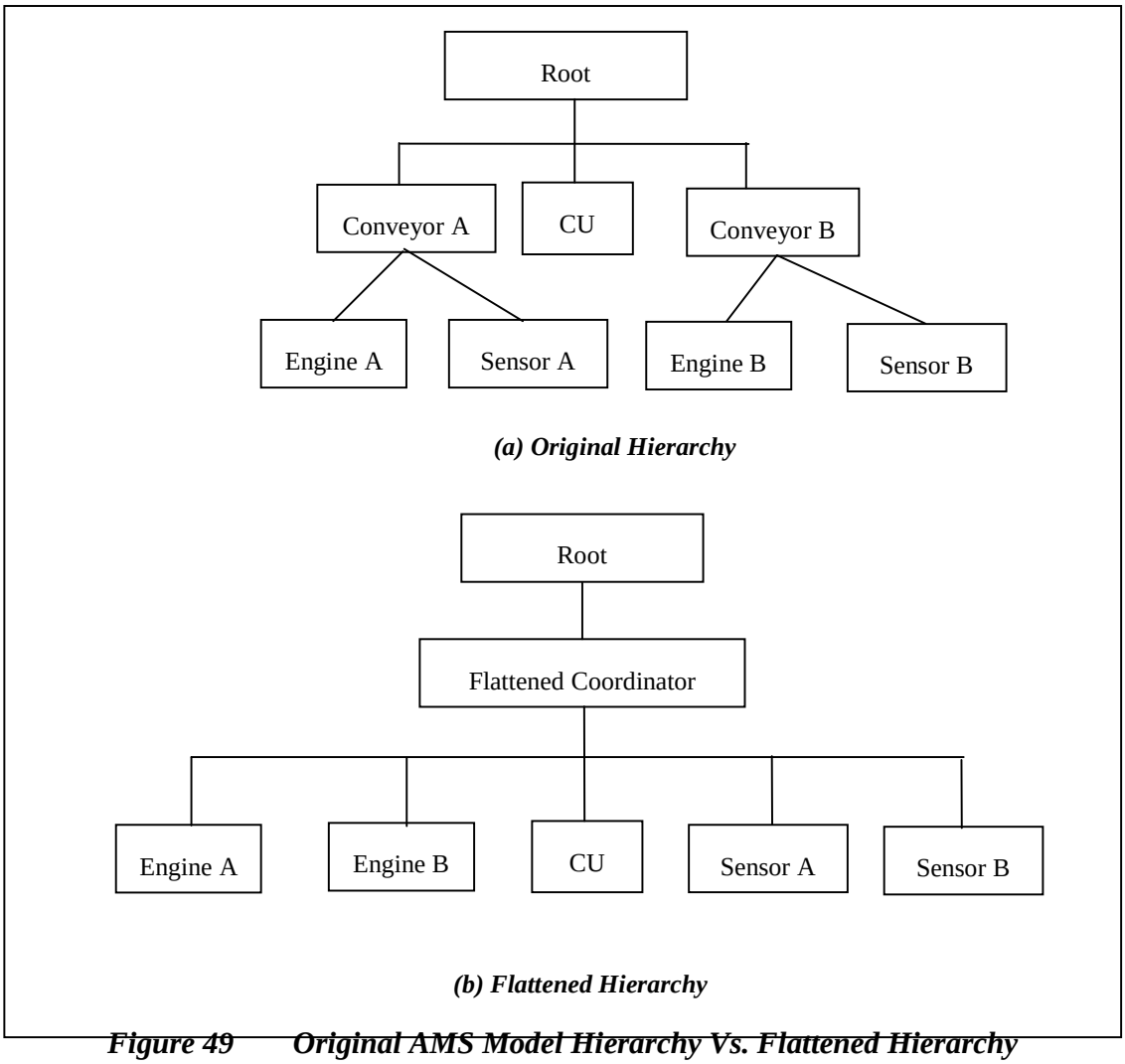

Figure 49 Original AMS Model Hierarchy Vs. Flattened Hierarchy

Based on the theory we developed in section 3.5, the theoretical improvement ratio is 29%, comparing to the experimental result of 33% (Table 7 shows the calculations).

Theoretical Result	Experimental Result
$R = 1 – (P_f / P_o) = 1 – 5 / 7 = 29\%$	$(385 – 257) / 385 = 33\%$

Table 7 Theoretical Vs. Experimental Performance Improvement Ratio of Flattened Coordinator Technique

There is a 12% difference between the two results. This disagreement is resulted from the bias of the data sample collected by the experiment. This experiment only ran a small simulation of processing two external events. When larger simulations were run, the experimental results tended to agree more with that of the theoretical.

5.2.3 Execution of Confluent Functions

Confluent functions are introduced by the P-DEVS formalism to resolve the conflict when, in an atomic model, the internal transition and the external transition happen at the same time. The confluent function is called to break the tie. The confluent functions feature is a major difference between the parallel CD++ and the non-parallel version. In the non-parallel CD++, the internal transition is always executed first to break the tie. The confluent function, however, gives the modeller the control to define the conflict resolution.

As an experiment, a confluent function is defined in the Controller Unit (CU). The external transition function in the CU, as shown in Figure 50, handles the incoming events from the scheduler and the signals from the sensor controllers in the conveyors. The internal transition function sets the CU model's internal state variable. The variable is called "`button_enabled`", which has impact on the logic of the external transition function. If the CU detects that conveyers are still transporting the products, its `button_enabled` variable is set to false. As a result, the CU's external transition function ignores any external events coming through the buttons input port. The `button_enabled` variable is set to true by the internal transition function when the conveyor delivers the product to its destination station. By then, the CU can start to handle the events coming through the buttons input port again.

When the product reaches a station, the sensor controller of the conveyor sends a signal to the CU indicating the current product position. The CU's external transition function handles this external event, and based on the product position information, the CU make decisions to control the conveyor's engine (e.g., continue moving or stopping). When this external transition time is elapsed, the internal transition function will be invoked to set the `button_enabled` value. If the product has reached the destination, `button_enabled` will be set to true. Otherwise, it will remain false.

```
Model &ECU::confluentFunction( const InternalMessage &msg, const
MessageBag &msgbag ){
  internalFunction( msg );

  MessageBag::iterator cursor = msgbag.begin();
  for( ; cursor != msgbag.end(); cursor++ )  {
    if ( ((*cursor)->port() == b2A) && (cur_station_A == 21) )    {
      led2 = true;
      req_station_A = 21;
      direction_A = 0;          //stop engine!!!
      holdIn(Atomic::active, Time::Zero);
    }else{
      externalFunction( *(( ExternalMessage* )( *cursor )) );
    }//if-else
  }//for

  return *this;
}//ECU::confluentFunction
```

Figure 50 Confluent Function of the Controller Unit

Conflicts may rise when a button is pressed at the same time when the internal transition function should also be invoked (i.e., t_n = 0). Figure 51 is an example of this situation.

Event time	*Associated deadline*	*input port*	*associated output port*	*Value*
00:00:02:100	00:00:05:300	Btn3A	bellA	1
00:00:03:550	00:00:05:300	Btn2A	bellA	1

Figure 51 A schedule events file that can cause conflicts

In the initial state, the product is placed at station 1. At the time 00:00:02:100, the button 3A is pressed indicating that the product needs to be transported to station 3. At the time 00:00:03:550, the button 2A is pressed. Figure 52 is the non-parallel CD++ simulation output of these 2 events.

actual output time (physical or wall-clock time)	*Associated deadline*	*Result*	*output port*	*value*
00:00:02:300		**No deadline**	Led3	1
00:00:02:300		**No deadline**	dirn_disp_a	1
00:00:03:550		**No deadline**	stn_disp_a	21
00:00:04:550		**No deadline**	stn_disp_a	31
00:00:04:550		**No deadline**	dirn_disp_a	0
00:00:04:550		**No deadline**	Led3	0
00:00:04:550	00:00:05:300	**Succeeded**	Bell_A	1

Figure 52 Output results generated by non-parallel CD++

Figure 52 implies that, the time 00:00:03:550, the CU's output function was called to make the display controller display the product position via the output port stn_disp_a.

Since the output function is always called right before the internal transition function is invoked, and also since that the hold in time of the display controllers' external transition function is zero, it follows that the CU's internal transition function was scheduled to be invoked at 00:00:03:550, which is the same time of the 2nd external event in the event file. This leads to a conflict. The non-parallel CD++ conflict resolution is to let the internal transition function be invoked first. As a result, the 2nd external event, i.e., the press of button 2A, is ignored because the `button_enabled` variable was set to false.

The modeller wants to change this conflict resolution behaviour, he or she must use parallel CD++ simulator and define a confluent function. Figure 53 captures the simulation results generated by the parallel CD++ simulator.

actual output time (physical or wall-clock time)	***Associated deadline***	***result***	***output port***	***value***
00:00:02:300		**No deadline**	Led3	1
00:00:02:300		**No deadline**	dirn_disp_a	1
00:00:03:550		**No deadline**	stn_disp_a	21
00:00:03:550		**No deadline**	Led2	1
00:00:03:550		**No deadline**	dirn_disp_a	0
00:00:03:550		**No deadline**	Led2	0
00:00:03:550		**No deadline**	Led3	0
00:00:03:550	00:00:05:300	**Succeeded**	Bell_A	1

Figure 53 Output results generated by parallel CD++

Figure 53 shows that the product on conveyor A stops at station 2 at the time 00:00:03:550. This is a result of the execution of the confluent function. That is, the CU's confluent function decides to stop the engine when the product reaching station 2 and the press of button 2A happens at the same time.

Chapter 6 Conclusions

This dissertation proposes a novel Modelling and Simulation-based development methodology for Real-time Embedded Systems. The motivation behind proposing this new framework is that the current state-of-the-art design methods for RTES are inadequate for providing a consistent and unified design framework throughout the entire development lifecycle, as well as failing to provide a formal product verification strategy. The proposed framework addresses these issues. The proposed methodology consists of modelling, model verification, and incremental model replacement phase. This new development cycle provides a consistent toolkits and terminology among analysis, design, implementation, and test. For instance, early DEVS models created in the modelling phase will not be abandoned but directly reused in the model verification and model replacement phase.

The creation of E-CD++ is a necessary and important step towards the realization of the proposed methodology. E-CD++ supports the RT-DEVS formalism by implementing P-DEVS and the Time Interval Function. We showed that the RT-DEVS formalism is adequate to model RTES. Consequently, DEVS model configurations can be formally verified against the target system's specifications. Meanwhile, E-CD++ also implemented a graphical model loader to support the GGAD graphical notation. The AMS experiment, shows that defining a DEVS model using GGAD takes much less effort than that doing so in C++. The work also draws another conclusion that the Flattened Coordinator Technique improves E-CD++ performance. The AMS experiment shows that the technique improves the E-CD++ performance by 33%.

Finally, with the help of E-CD++, DEVS models can be executed directly in an embedded environment and can also interact directly with hardware surrogates and real-world events, which supports the seamless transition from the modelling phase to implementation phase. We illustrated this transition by showing how E-CD++ was used to design and develop the AMS.

6.1 Future work

E-CD++ is a newly created software. There are still a lot of areas where E-CD++ can be improved or extended. We list two topics of interest that we think worthwhile to study.

The first topic would be the *Parallel Execution of Multiple Flattened Coordinators*. In theory, the Flattened Coordinator Technique can improve the performance of any DEVS model hierarchy. In implementation, however, there is a scalability issue, because time delay exists in accessing and retrieving Atomic model objects from the Atomic Model Database. As the size of the model hierarchy grows, so is the database. Currently this database is implemented as an ordered list on ta(s). When the database size grows too large (e.g., containing thousands of Atomic models), the time delay incurred in accessing the database will eventually outnumber the performance improvements gained from the technique itself. One solution to solve this problem is to partition the Atomic models into multiple smaller databases and create multiple Flattened Coordinators. And ideally, these Flattened Coordinators can run in their own task spaces, so that they can run in parallel and, therefore, maximize the performance.

We may also be interested in *Adding Confluent Function Support in the GGAD Language*. The current GGAD language does not support confluent functions. To add confluent functions, we first need to make GGAD front-end changes to add confluent function's grammar definition to the GGAD Parser. We also need to make back-end changes. We need to create a new subclass, say `GgadConfluentFunction`, under the `GgadSyntaxNode` class to represent the confluent function in the Syntax Tree. Then we capture the behaviour of confluent functions in the method `GgadConfluentFunction::evaluate()`. The major back-end work is the implementation of this function.

Chapter 7 References

[Alu03] Alur, R.; Thao Dang; Esposito, J.; Yerang Hur; Ivancic, F.; Kumar, V.; Mishra, P. “Hierarchical modelling and analysis of embedded systems”. *Proceedings of the IEEE*. Volume 91, Issue 1, Jan. 2003 Page: 11 – 28.

[CEP99] Cortes, L. A.; Eles, P.; Peng, Z. “A Survey on Hardware/Software Codesign Representation Models”. *SAVE Project Report, Linkoping University*. Sweden. June 1999.

[Cho94a] Chow, A; Zeigler, B. “Parallel DEVS: A parallel, hierarchical, modular modelling formalism”. *Proceedings of the 26th conference on Winter simulation, Orlando, Florida, USA*. 1994, Page(s): 716-722.

[Cho94b] Chow, A.; Kim D.; Zeigler, B. “Abstract Simulator for the parallel DEVS formalism”. *Proceedings of the 5th annual conference on AI, Simulation, and Planning in High Autonomy Systems, Gainesville, Florida, USA*. December 1994, Page(s): 157:163.

[CK01] Cho, S. M.; Kim, T. G. “Real time simulation framework for RT-DEVS models”. *Transactions of the Society for Computer Simulation International archive.* Volume 18, Issue 4, December 2001, Page(s): 203 – 215.

[Ern98] Ernst, R. “Codesign of embedded systems: status and trends”. *IEEE Design & Test of Computers*. Apr-Jun 1998. Volume: 15, Issue: 2, Page(s): 45-54.

[Gli04] Glinsky, E. “New Techniques for Parallel Simulation of DEVS and Cell-DEVS Models in CD++”. *M. A. Sc. Dissertation, Carleton University, Canada*. 2004.

[Gup02] Gupta, P. “Hardware-software codesign”. *Potentials, IEEE*. Volume 20, Issue 5, Dec 2001-Jan 2002, Page(s):31 – 32.

[Har96] Harel, D.; Naamad, A. “The StateMate Secmantics of StateCharts”. *ACM Transactions on Software Engineering Methodology*. October 1996, Page(s): 293-333.

[HS04] Huang, D., H.S. Sarjoughian. "Software and Simulation Modelling for Real-time Software-intensive System". *The 8th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Budapest, Hungary. October 2004, Page(s): 196-203.

[HSKP97] Hong, J. S.; Song, H. S.; Kim, T. G.; Park, K. H. “A real-time Discrete Event System Specification formalism for seamless real-time software development”. *Discrete Event Dynamic Systems.* Volume 7, Issue 4, 1997, Page(s):355-375.

[HZC01] Hu, X.; Zeigler, B.P.; Couretas, J. “DEVS-on-a-Chip: Implementing DEVS in Real-time Java on a Tiny Internet Interface for Scalable Factory Automation”. *Proceedings of the 2001 IEEE International conference on Systems, Man, and Cybernetics Conference, Tucson, AZ, USA*. Volume 5, 2001, Page(s): 3051-3056.

[JRH03] Jones, E. D.; Roberts, R. S.; Hsia, T. C. S.; “STOMP: A Software Architecture for the Design and Simulation of UAV-based Sensor Networks”. *Proceedings of the 2003*

IEEE International Conference on Robotics & Automation. Taipei, Taiwan. September 14-19, 2003.

[Kim00] Kim, K.; Kang W.; Sagong, B.; Seo, H. “Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One”. *Proceedings of the 33rd Annual Simulation Symposium, Washington DC, USA*. 2000.

[LDNA03] Ledeczi, A.; Davis, J.; Neema, S.; Agrawal, A. “Modelling methodology for integrated simulation of embedded systems”. *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, Volume 13 Issue 1, January 2003.

[LG05] Yvan Labiche; G. Wainer. "Towards the Verification and Validation of DEVS Models". *Proceedings of the 1st Open International Conference on Modelling & Simulation, Clermont-Ferrand, France*. 2005.

[LPW03] Li, L.; Pearce, T.; Wainer, G. “Interfacing Real-time DEVS models with a DSP platform”. *Proceedings of the Industrial Simulation Symposium, Valencia, Spain*. 2003.

[RCL00] Reyneri, LM; Chiaberge, M; Lavagno, L.; “Simulink-based HW/SW codesign of embedded neuro-fuzzy systems”. *International Journal of Neural Systems*. Volume 10, Issue 3, June 2000, Page(s): 211-226.

[SER00] Schulz, S.; Ewing, T.C.; Rozenblit, J.W. “Discrete event system specification (DEVS) and StateMate StateCharts equivalence for embedded systems modelling”. *Proceedings of the Seventh IEEE International Conference and Workshop*. Volume 3, Issue7, April 2000, Page(s):308 – 316.

[SK05] Song, H. S.; Kim, T. G. “Application of Real-Time DEVS to Analysis of Safety-Critical Embedded Control Systems: Railroad Crossing Control Example”. *Simulations.* Volume 81, Number 2 2005.

[SLS00] 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 Page(s):14 – 27.

[Sta88] Stankovic, J. “Misconceptions about real time computing: A serious problem for next generation systems”. *IEEE Computer*. Volume 21, Issue 10, October 1988, Page(s): 10-19.

[Sta96] Stankovic, J. “Strategic Directions in Real-Time and Embedded Systems”. *ACM Computing Surveys*. 50th Anniversary Issue, Volume 28, Issue 4, December 1996, Page(s): 751-763.

[Wai02] Wainer, G. “CD++: a toolkit to develop DEVS models”. *Software – practice and Experience*. Volume 32, 2002, Page(s): 1261 – 1306.

[Wai98] "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation". G. Wainer, N. Giambiasi. *Simulation*, Vol. 71, No. 1. January 2001.

[Zei76] Zeigler, B. *Theory of Modelling and Simulation.* Wiley. 1976.

[ZKP00] Zeigler, B.; Kim, T.; Praehofer, H. Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. 2nd Edition. Academic Press. 2000.

Appendix A E-CD++ System Architecture

A.1.The Client-Server System Architecture

The Ampro LB700^{TM} board is chosen as the SBC for E-CD++, although nothing prevents the E-CD++ from running on other hardware platforms. The LB700 board has a 700MHz Intel x86 CPU, 256 megabytes of RAM and 2 Ethernet ports. It has no hard disk. All the software images running on the board is loaded in the memory during run time.

A customized Linux 2.4 kernel is used as the operating system (OS) for E-CD++. The OS supports NFS over Ethernet and ramdisk, yet the memory swapping is disabled (due to the lack of the hard disk on the SBC).

Figure 54 illustrates the system architecture of the E-CD++ toolkit. The E-CD++ system architecture adopts the client-server computing model. The SBC interacting with the real world is the client, and the host simulation workstation is the server. The client and the server are connected via Ethernet ports. The Bootrom firmware image on the SBC is configured to be able to transfer the Linux kernel image from the server over the Ethernet and load it into the SBC's memory, when the SBC is boot up.

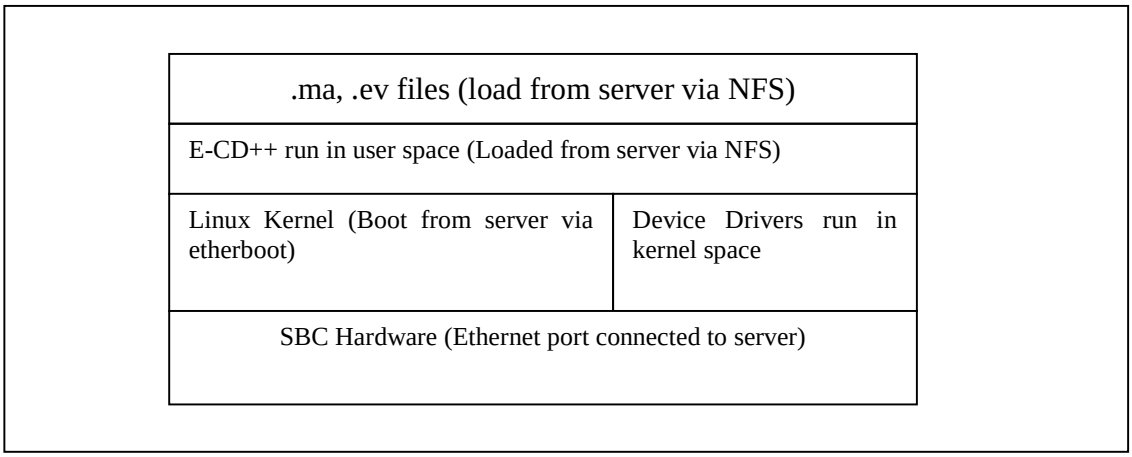

Figure 54 E-CD++ Software Architecture

A.2. The SBC Booting Sequence

The following is the SBC's booting sequence:

1. The Bootrom firmware on the SBC is Etherboot – a less than 16K bootable image stored in the Ethernet socket on the SBC. The Etherboot image is configured to load the customized Linux kernel that is stored on the server into the SBC's memory via the dhcp and the tftp protocol.

2. Once the kernel has been loaded into memory, it will begin executing.

3. The kernel will initialize the entire system and all of the peripherals on the SBC.

4. During the kernel loading process, a ramdisk image will also be loaded into memory. A kernel command line argument of **root=/dev/ram0** tells the kernel to mount the image as the root directory.

5. When the kernel is finished booting, it is instructed to launch the **/linuxrc** script. This is achieved passing **init=/linuxrc** on the kernel command line.

6. The **/linuxrc** script begins by loading the correct Ethernet driver module into the kernel space.

7. A small DHCP client called **dhclient** will then be run, to make another query from the DHCP server. This separate user-space query is necessary, because we need more information than the Etherboot retrieved with the first dhcp query.

8. When **dhclient** gets a reply from the server, it will run the **/etc/dhclient-script** file, which will take the information retrieved, and configure the eth0 interface.

9. Upto this point, the root filesystem has been a ram disk. Now, the /linuxrc script will mount a new root filesystem via NFS. The directory that is exported from the server

is **/tftpboot/henryRoot**. It can't just mount the new filesystem as /. It must first mount it as /mnt. Then, it will do a **pivot_root**. pivot_root will swap the current root filesystem for a new filesystem. When it completes, the NFS filesystem will be mounted on /, and the old root filesystem will be mounted on /oldroot.

10. Once the mounting and pivoting of the new root filesystem is complete, we are done with the /linuxrc shell script and we need to invoke the real **/sbin/init** program.

11. Init will read the file and begin setting up the workstation environment.

12. One of the first items in the inittab file is the **rc.sysinit** command that will be run while the workstation is in the '**sysinit**' state.

13. The rc.sysinit script will create a 1mb ramdisk to contain all of the things that need to be written to or modified in any way.

14. The ramdisk will be mounted as the directory. Any files that need to be written will actually exist in the directory, and there are symbolic links pointing to these files.

15. The filesystem is mounted.

16. Memory swapping is disabled by rc.sysinit.

17. The **loopback** network interface is configured. This is the networking interface that has *127.0.0.1* as its IP address.

18. Local applications are enabled, for E-CD++ runs in user space. The **/usr/local/bin** directory is mounted. That is location where E-CD++ is installed.

19. Several directories are created in the file system for holding some of the transient files that are needed while the system is running. Directories such as:

- /tmp/compiled
- /tmp/var
- /tmp/var/run
- /tmp/var/log
- /tmp/var/lock
- /tmp/var/lock/subsys

will all be created.

20. Once the rc.sysinit script is finished, control returns back to the /sbin/init program, which will change the runlevel from **sysinit** to **5**. This will cause any of the entries in the file to be executed.

Appendix B Grammar for GGAD Models

B.1. Context-free Grammar for GGAD models

```
rule 1    Ggad -> ModelName GGADT_EOL GgadRules
rule 2    ModelName -> GGADT_LBRACKET GGADT_ID
GGADT_RBRACKET
rule 3    GgadRules -> GgadRule GGADT_EOL GgadRules
rule 4    GgadRules -> GgadRule GGADT_EOL
rule 5    GgadRules -> GgadRule
rule 6    GgadRule -> InDecl
rule 7    GgadRule -> OutDecl
rule 8    GgadRule -> StateDecl
rule 9    GgadRule -> VarDecl
rule 10   GgadRule -> StateDef
rule 11   GgadRule -> InitialState
rule 12   GgadRule -> IntDef
rule 13   GgadRule -> ExtDef
rule 14   GgadRule -> VarDef
rule 15   InDecl -> GGADT_IN GGADT_COLON PortInIdList
rule 16   OutDecl -> GGADT_OUT GGADT_COLON PortOutIdList
rule 17   VarDecl -> GGADT_VAR GGADT_COLON VarIdList
rule 18   VarDef -> GGADT_VARIABLEID GGADT_COLON
GGADT_CONSTANT
rule 19   StateDecl -> GGADT_STATE GGADT_COLON StateIdList
rule 20   StateDef -> GGADT_STATEID GGADT_COLON
GGADT_TIME_CONSTANT
rule 21   StateDef -> GGADT_STATEID GGADT_COLON
GGADT_INFINITE
rule 22   InitialState -> GGADT_INITIAL GGADT_COLON
GGADT_STATEID
rule 23   IntDef -> GGADT_INT GGADT_COLON GGADT_STATEID
GGADT_STATEID PortValueOutList Actions
rule 24   PortValueOutList -> GGADT_PORTID GGADT_OUTPUT
Expression PortValueOutList
rule 25   PortValueOutList -> /* empty */
rule 26   ExtDef -> GGADT_EXT GGADT_COLON GGADT_STATEID
GGADT_STATEID Expresion GGADT_INPUT GGADT_CONSTANT Actions
rule 27   Expresion -> FunctionCall
rule 28   Expresion -> GGADT_PORTID
rule 29   Expresion -> GGADT_VARIABLEID
rule 30   Expresion -> GGADT_CONSTANT
rule 31   FunctionCall -> GGADT_FUNCTIONID GGADT_LPAR
ActualParamList GGADT_RPAR
```

```
rule 32   ActualParamList -> ActualParameter
rule 33   ActualParamList -> ActualParameter GGADT_COMMA
ActualParamList
rule 34   ActualParameter -> GGADT_CONSTANT
rule 35   ActualParameter -> GGADT_VARIABLEID
rule 36   ActualParameter -> GGADT_PORTID
rule 37   StateIdList -> StateIdList GGADT_ID
rule 38   StateIdList -> GGADT_ID
rule 39   PortInIdList -> PortInIdList GGADT_ID
rule 40   PortInIdList -> GGADT_ID
rule 41   PortOutIdList -> PortOutIdList GGADT_ID
rule 42   PortOutIdList -> GGADT_ID
rule 43   VarIdList -> VarIdList GGADT_ID
rule 44   VarIdList -> GGADT_ID
rule 45   Actions -> GGADT_BEGIN ActionList GGADT_END
rule 46   Actions ->/* empty */
rule 47   ActionList -> Action GGADT_SEMICOLON
rule 48   ActionList -> ActionList Action GGADT_SEMICOLON
rule 49   Action -> GGADT_VARIABLEID GGADT_ASSIGNMENT
Expresion
```

B.2. Tokens:

```
GGADT_CONSTANT
GGADT_IN                reserved word "in"
GGADT_OUT               reserved word "out"
GGADT_STATE             reserved word "state"
GGADT_INITIAL           reserved word "initial"
GGADT_ID                an identifier
GGADT_STATEID           a state identifier
GGADT_PORTID            a port identifier
GGADT_FUNCTIONID        a function identifier
GGADT_VARIABLEID        a variable identifier
GGADT_INT               reserved word "int"
GGADT_EXT               reserved word "ext"
GGADT_VAR               reserved word "var"
GGADT_CONSTANT          integer o real constant
GGADT_TIME_CONSTANT     time constant in cd++ format hh:mm:ss:nn
GGADT_INFINITE          reserved word "infinite"
GGADT_COLON             ":"
GGADT_EOL               end of line character
GGADT_OUTPUT            output operator "!"
GGADT_INPUT             input operator "?"
GGADT_LPAR              "("
GGADT_RPAR              ")"
GGADT_LBRACKET          "["
GGADT_RBRACKET          "]"
GGADT_COMMA             ","
GGADT_BEGIN             "{"
GGADT_END               "}"
GGADT_SEMICOLON         ";"
GGADT_ASSIGNMENT        "="
```

B.3. GGAD Built-in Functions

```
addFunction( "value", new GgadFuncValue() );
addFunction( "add", new GgadFuncAdd() );
addFunction( "minus", new GgadFuncMinus() );
addFunction( "multiply", new GgadFuncMultiply() );
addFunction( "divide", new GgadFuncDivide() );
addFunction( "pow", new GgadFuncPow() );
addFunction( "between", new GgadFuncBetween() );
addFunction( "compare", new GgadFuncCompare() );
addFunction( "any", new GgadFuncAny() );
addFunction( "pi", new GgadFuncPi() );
addFunction( "equal", new GgadFuncEqual() );
addFunction( "notequal", new GgadFuncNotEqual() );
addFunction( "and", new GgadFuncAnd() );
addFunction( "or", new GgadFuncOr() );
addFunction( "not", new GgadFuncNot() );
addFunction( "rand", new GgadFuncRand() );
addFunction( "less", new GgadFuncLess() );
addFunction( "greater", new GgadFuncGreater() );
addFunction( "greaterequal", new GgadFuncGreaterEqual() );
```