Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

2002

Definición de Simulación en Tiempo Real en CD++

Tesis de Licenciatura en Ciencias de la Computación

Autor

Ezequiel J. Glinsky

Director

Dr. Gabriel Wainer

Table of Contents

ABSTRACT

The CD++ toolkit was developed in order to implement the theoretical concepts specified by the DEVS formalism. The tool allows the execution of both DEVS and Cell-DEVS models. In this work, we present a synthetic model generator that produces DEVS models similar to those that exist in the real world. A thorough testing has been carried out using the different simulation techniques provided in the toolkit, which employ a virtual time approach. This work presents the definition and implementation of a real time simulator. In such simulations, events must be handled timely and time constraints can be stated and validated accordingly. The new simulation technique allows the interaction between the model and its surrounding environment. Additionally, a non-hierarchical simulation approach is presented and introduced to CD++ in order to reduce the communication overhead. The experiments showed that the new flattened simulation technique is more efficient than the hierarchical one.

1. INTRODUCTION

1.1 Introduction to simulation

Simulation is a powerful tool for analyzing and understanding a wide variety of complex systems. The simulation process begins with a problem to be solved or understood, such as urban traffic, network performance or the spread of a virus through a group of cells. By observing the **real system**, different entities are identified. A **model** is an abstract representation of such system that is constructed accordingly. The execution of the model is carried out by a **simulator**. The simulator consists of a computer system that executes the instructions of that model to generate its behavior. Finally, the obtained results are compared to those of the real system for validation. Usually, the modeler is interested in only a few aspects of the real system. Consequently, an **experimental frame** is defined to bound the scope of the model, composed of a limited set of circumstances under which the real system is being studied [Zei76, Zei00].

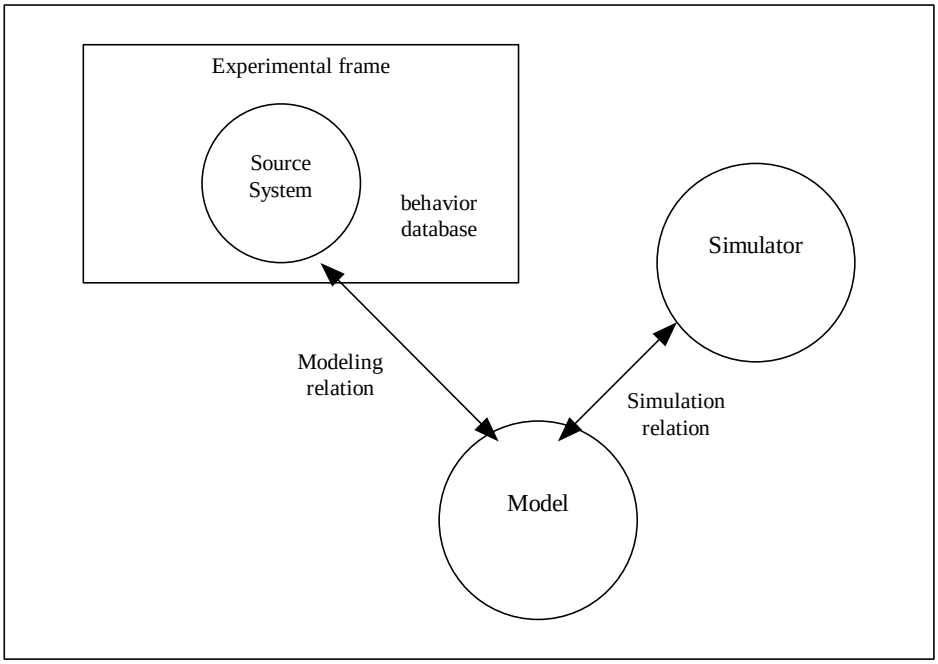


Figure 1: The basic entities and their relationships [Zei00]

The basic entities are linked by two relations [Zei00]:

- *modeling relation:* Links the real system and the model. It defines how well the model represents the system or entity being modeled. Generally, a model can be considered valid if the data generated by the model agrees with the data produced by the real system in the experimental frame of interest.

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

Different formalisms exist to model and simulate real and artificial systems. Among these, **DEVS** [Zei76, Zei00] is a widely used formalism, which is described in the next section.

1.2 Introduction to the DEVS formalism

Systems whose variables are discrete and where time advance is continuous are known as **DEDS** (**Discrete Event Dynamic Systems**), as opposed to **CVDS** (**Continuous Variable Dynamic Systems**) described by differential equations. Simulation mechanisms for DEDS systems assume that changes of state will take place at discrete points of time, upon the occurrence of an event. Formally, an **event** is defined as a change of state that occurs at a specific point of time $t_i \in \mathbf{R}$.

DEVS (Discrete EVents Systems Specification) [Zei76, Zei00], a formalism for modeling and simulating DEDS systems, defines a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. It allows hierarchical decomposition of the model by defining a way to couple existing DEVS models.

A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components. An *atomic* model 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 shows the description of states and variables in DEVS models:

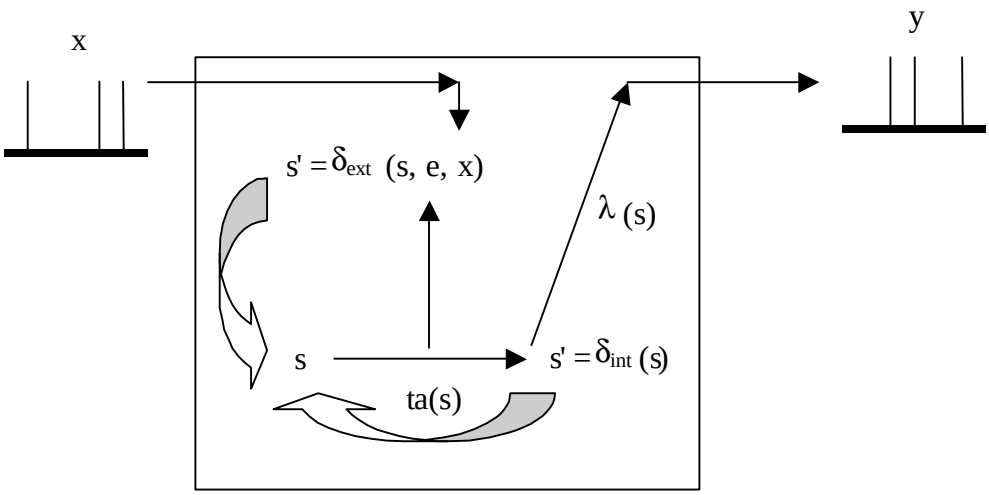


Figure 2: DEVS Semantics

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

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

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* ∪ { *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*,

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

I_i is a subset of *D* ∪ { *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.

Note that multiple components can be scheduled for an internal transition at the same time in a coupled component, and therefore ambiguity may arise. If the first component to execute its internal transition produces an output that maps to an external event for another component that is already scheduled for an internal transition, then it is not clear which transition this second component should execute first. 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*. By the *select* function, the DEVS formalism solves this ambiguity. 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), 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.

1.3 Introduction to the Cell-DEVS formalism

1.3.1 Cellular Automata

Cellular Automata are used to describe real systems that can be represented as a cell space. A cellular automaton is an infinite regular *n*-dimensional lattice whose cells can take one finite value. The states in the lattice are updated according to a local rule in a simultaneous and synchronous way. The cell states change in discrete time steps as dictated by a local transition function using the present cell state and a finite set of nearby cells (called the neighborhood of the cell).

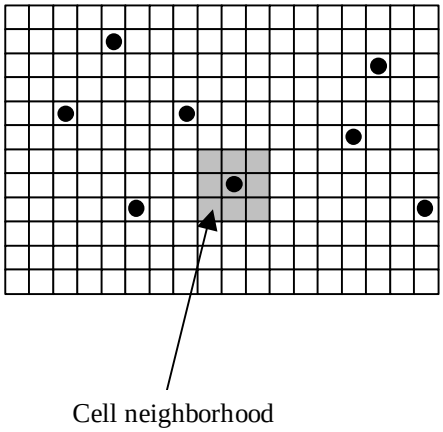


Figure 3: Sketch of a cellular automaton [Wai00]

The Timed Cell-DEVS formalism [Wai98] uses the DEVS paradigm to define a cell space where each cell is defined as a DEVS atomic model. As a result, it is possible to build discrete event cell spaces improving their definition by making the timing specification more expressive.

1.3.2 The Timed Cell-DEVS formalism

Cell-DEVS defines a cell as DEVS atomic model. A Cell-DEVS atomic model is defined by [Wai98]:

$$TDC = < X, Y, I, S, \theta , N, d, \delta_{int}, \delta_{ext}, \tau , \lambda , D >$$

where

X is a set of external input events;

Y is a set of external output events;

I represents the model's modular interface;

S is the set of sequential states for the cell;

Θ is the cell state definition;

N is the set of states for the input events;

d is the delay for the cell;

δ_{int} is the internal transition function;

δ_{ext} is the external transition function;

τ is the local computation function;

λ is the output function; and

D is the state duration function.

A cell uses a set of input values N to compute its future state, which is obtained by applying the local computation function τ. A delay function is associated with each cell, deferring the output of the new state to the neighbor cells. This activation of the local computation is carried by the δ_{ext} function.

After the basic behavior for a cell is defined, a complete cell space can be constructed by building a coupled Cell-DEVS model:

$$GCC = < X_{list}, Y_{list}, I, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z, select >$$

where

X_{list} is the input coupling list;

Y_{list} is the output coupling list;

I represents the definition of the interface for the modular model;

X is the set of external input events;

Y is the set of external output events;

n is the dimension of the cell space;

$\{t_1,...,t_n\}$ is the number of cells in each of the dimensions;

N is the neighborhood set;

C is the cell space;

B is the set of border cells;

Z is the translation function; and

select is the tie-breaking function for simultaneous events.

This specification defines a coupled model composed of an array of atomic cells. Each cell is connected to the cells defined in its neighborhood. Nevertheless, as the cell space is finite, either the borders are provided with a different neighborhood than the rest of the space, or they are *wrapped* (cells in one border are connected with those in the opposite one). Finally, the *Z* function defines the internal and external coupling of cells in the model. This function translates the outputs of m-th output port in cell C_{ij} into values for the m-th input port of cell C_{kl} . Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood. The select function serves the same purpose as in the original DEVS models: to tiebreak among imminent components.

1.4 Introduction to the CD++ toolkit

CD++ implements DEVS and Cell-DEVS theory, allowing the definition and simulation of models using the specification described in the previous sections [Rod99, Wai01]. The tool was built as a hierarchy of classes in C++, each of them corresponds to a simulation entity using the basic concepts defined in [Zei76, Zei00].

Two basic abstract classes exist: *Model* and *Processor*. The former is used to represent the behavior of the atomic and coupled models, while the latter implements the simulation mechanisms. *Figure 4* shows the CD++ class hierarchy.

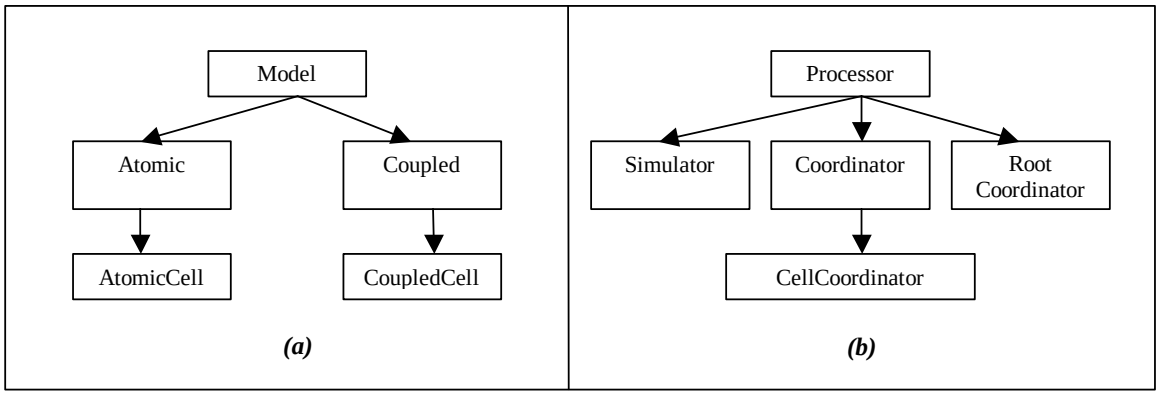


Figure 4: CD++ (a) Model hierarchy, (b) Processor hierarchy

The *Atomic* class implements the behavior of an atomic component. The *Coupled* class implements the mechanisms of a coupled model. For cellular models, special atomic models are used to represent the cells. To do so, *AtomicCell* and *CoupledCell* are defined as subclasses of *Atomic* and *Coupled* respectively. *AtomicCell class* extends the behavior of the atomic models, to define the functionality of the cell space. In contrast, *CoupledCell* handles a group of atomic cells.

A *simulator* object manages an associated atomic object, handling the execution of its δ_{int}, δ_{ext} and λ(s) functions. A *coordinator* object manages an associated coupled object. Only one *root coordinator* exists in a simulation. It manages global aspects of the simulation. It is involved with the topmost-coupled component, which has the highest level in the model hierarchy. Moreover, the *root coordinator* maintains the global time, and it starts and stops the simulation process. Lastly, it receives the output results that must be sent to the environment.

The simulation process is message driven; it is based on the message exchange among processors. Each message contains information to identify the *sender* and the *receiver*. A *time-stamp* for the message and an associated *value* are also included in the packet. Two main categories of messages exist: **synchronization** and **content** messages. These categories are consisted of several types of messages.

Synchronization messages:

@	*Collect message*
*******	*Internal message*
done	*Done message*

Content messages:

q	*External message*
y	*Output message*

In addition, a *processor* has internal variables to keep the time of the simulation:

t_L	*Time of last transition*
t_N	*Time of next transition*

and a *bag* to store external messages.

The tool provides a specification language that allows describing coupling of models, initial values and external input events. Additionally, atomic models are developed under C++, which provides a great flexibility and computing power to the modeler. Each new atomic model must inherit from the *Atomic* class in order to extend their basic behavior.

Lastly, for Cell-DEVS model execution, CD++ allows defining size and structure of the cell space and its connection with other existing DEVS models, type of delay, neighborhood, border and initial state for each cell.

This work is organized as follows. Chapter 2 introduces the simulation techniques available in the CD++ toolkit. Furthermore, a synthetic model generator is developed and presented. Finally, a performance analysis of the simulation techniques is provided. Chapter 3 introduces a real-time extension to the CD++ toolkit, while Chapter 4 presents the testing of such simulator. In order to provide better performance, a flattened simulator is introduced in Chapter 5. Benchmark experiments are carried out using this new flattened approach in the sixth chapter. Chapter 7 provides conclusions about this work. An appendix presents tools developed to support the CD++ project.

2. PERFORMANCE ANALYSIS OF DIFFERENT SIMULATION TECHNIQUES

This chapter presents the available simulation techniques in the CD++ toolkit. In addition, a new synthetic model generator is presented. Furthermore, a thorough performance analysis is provided in order to characterize the overhead incurred by each technique.

2.1 Description of the available simulation techniques in CD++

Currently, the CD++ toolkit supports two different approaches to simulate DEVS and Cell-DEVS models.

The available distributions are the *original stand-alone version* [Rod99] and the *parallel version* [Tro01a]. The former is a one-processor simulation technique. The latter allows the execution of simulations on a distributed environment. In this section, both techniques are explained with more detail.

2.1.1 Original stand-alone simulator

The *original stand-alone simulator* [Rod99] can be used when the simulation is executed in only one processor. This is the simplest version of the CD++ toolkit and simulates both DEVS and Cell-DEVS models. It has been used on a variety of models including: traffic, forest fires, robot movement and watershed simulation [Ame00a] among others.

2.1.2 Parallel simulator

Eventually, the execution of more complex models requires a computing power that a stand-alone computer does not provide. Nevertheless, this computing power may be obtained by parallel and distributed systems.

Not only Cell-DEVS models, but also DEVS models may require this approach. The parallel version of the tool was developed using the Parallel DEVS [Cho94] and Parallel Cell-DEVS [Wai00], which are revisions of the DEVS and Cell-DEVS formalisms respectively.

When the parallel simulator [Tro01a] is invoked, synchronization between processes is needed. For flexibility, the parallel simulator was designed as a layered architecture application. The topmost layer implements the abstract simulator. The middle layer carries out synchronization between processes and the lowest layer is in charge of the communications between the CPUs. The middleware is provided by the Warped project [Mar97], which supports two different synchronization protocols. *TimeWarp kernel* provides the optimistic protocol. On the other hand, *NoTime kernel* implements an unsynchronized protocol. Both protocols are supported by the parallel version of the CD++ toolkit.

When distributed simulation is invoked, Warped uses MPI for the message passing. The complete layered architecture is shown below.

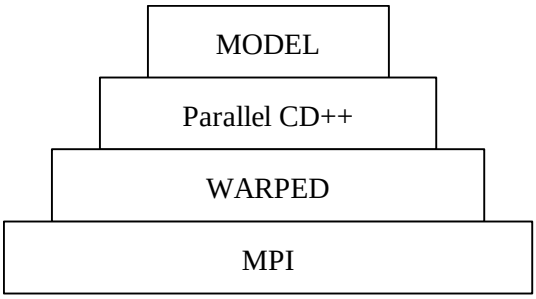


Figure 5: Parallel CD++ layered structure

The parallel CD++ version supports not only distributed, but also stand-alone execution. In the latter case, the MPI layer is not needed.

2.2 Synthetic model generator

The performance testing of a simulator tool is usually a very complex task. To make the analysis of the different DEVS simulation tools easier and more accurate, a synthetic experimental frame has been developed.

In order to perform a thorough study of the overheads incurred by each simulation technique, the synthetic model generator must be able produce a wide variety of models. The produced models must be similar to the ones that are studied in the real world.

To characterize a model, one should consider different aspects of its shape and behavior. Some of the most important characteristics are: *number of levels in the model hierarchy, number of atomic components, number of coupled components, total size, number of interconnections between components,* and *workload in internal and external transitions*.

The synthetic generator produces models of different shapes and behaviors using the following parameters:

- *model_type:* this parameter allows us to choose among different predefined interconnections between the model components. The amount of messages involved in a simulation is related to the number and type of links between the components.

- *depth:* determines the number of levels of the modeling hierarchy.

- *width:* determines the number of children each intermediate coupled component has. Along with depth, it establishes the size of the model.

- *#intdhrystones:* indicates the execution time to be consumed in the internal transition function, which simulates code to be executed.

- *#extdhrystones:* indicates the execution time to be consumed in the external transition function, which simulates code to be executed.

As stated in Chapter 1, an atomic model has two associated functions: the external transition and internal transition functions. The former executes whenever an external event arrives through an input port. The latter is executed before the model changes its state.

We used the Dhrystone benchmark [Wei84] to generate different workload in both transition functions. Dhrystone code is a synthetic benchmark intended to be representative for system (integer) programming. The

#intdhrystones and *#extdhrystones* parameters allow the execution of time-consuming code inside the internal and external transition, according to the number of milliseconds specified.

2.2.1 Type-1 models

This model type has a small number of interconnections between components. As a result, this characteristic allows the performance analysis in presence of a small number of messages exchanged.

2.2.1.1 Sample Type-1 model

The following is a sample Type-1 model generated with the tool. The width used here is three; hence, there are three components per level. The height used in this sample is four. *Figure 6* shows the top model, which is the first and topmost-coupled component described in the hierarchy.

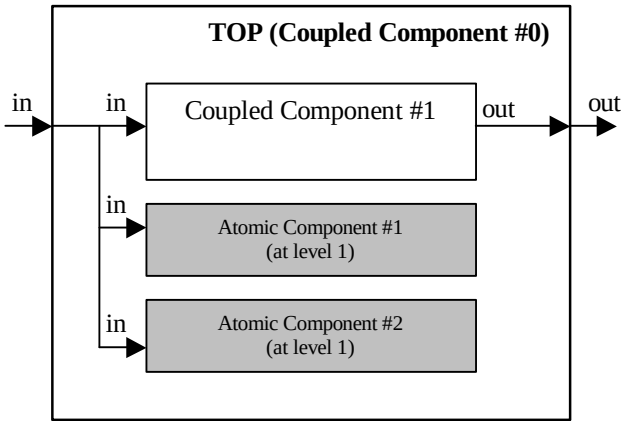


Figure 6: Top model (type 1)

The *arrows* indicate the existing input and output ports in each depicted model. *Boxes* denote the different components in the model. *Solid-white boxes* represent coupled components and *shaded-gray boxes* represent atomic components.

The *Top model* (*Coupled Component #0*) consists of one coupled component (labeled as *Coupled Component #1*) and two atomic ones (labeled as *Atomic Component #1* and *Atomic Component #2*) as shown above.

Coupled Component #1 is depicted below. It has the same internal structure as the *Top model* (*Coupled Component #0*) and therefore contains one coupled model (*Coupled Component #2*) and two atomic ones (*Atomic Component #3* and *Atomic Component #4*).

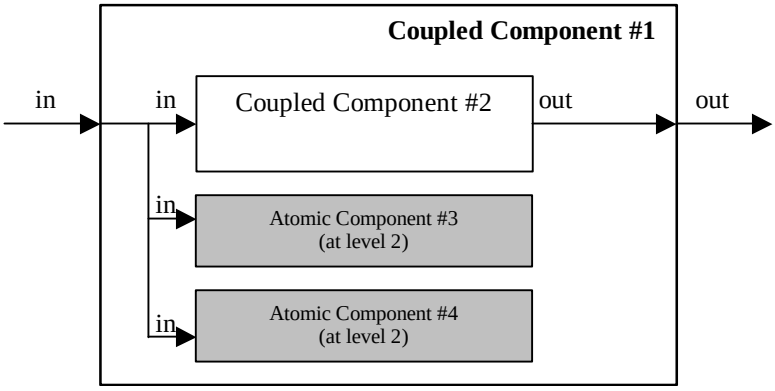


Figure 7: Coupled Component #1 (type 1)

Likewise, *Coupled Component #2* also repeats this structure, and accordingly contains *Coupled Component #3* and *Atomic Components #5* and *#6*.

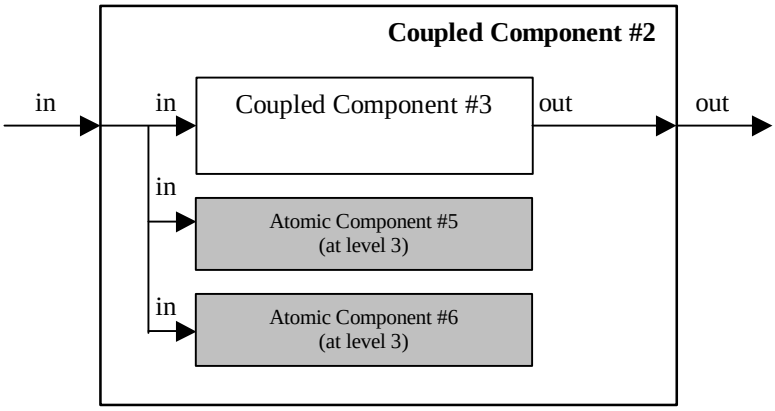


Figure 8: Coupled Component #2 (type 1)

Lastly, *Coupled Component #3* is simpler than those shown above. It has only one atomic child (#7) connected to its output port, regardless of the specified width.

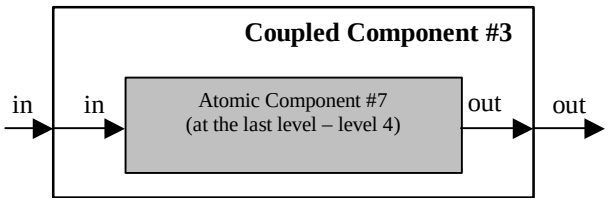


Figure 9: Coupled Component #3 (type 1)

Figure 9 shows a much more simple model because the chosen height was four for this model, and therefore this is the last coupled component in the hierarchy.

2.2.1.2 Characteristics of Type-1 models

To measure the time involved upon the reception of an external event, we have to take into account the number of internal and external functions being executed. This value is determined by the number of atomic components in the obtained model.

In general, given a specified *d* depth and *w* width, we end up having *d* coupled components with *w-1* atomic components inside each coupled one (except for the innermost coupled model in the hierarchy, which only has one atomic component).

Consequently, the total number of atomic components in a model generated by the tool is:

$$\textbf{\# Atomic Models = (width – 1) * (depth – 1) + 1}$$

In the above example, where *width* = 3 and *depth* = 4, we have 7 atomic models:

$$\textit{\# Atomic Models} = (3 - 1) * (4 - 1) + 1 = \mathbf{7}$$

Now, we can calculate how many atomic components a generated model has. Then, we also know the number of transition functions to be executed upon the reception of an external event.

In addition, all the atomic components spend a certain amount of time executing Dhrystone code in the external and internal transitions. Recall that this time is specified by the *#intdhrystones* and *#extdhrystones* parameters described before.

Using the previous data, now we can measure the time spent upon of the reception of an external event for Type-1 models. We must multiply the number of internal and external transitions to be executed by the amount of time spent in each transition function to obtain the total time needed to process a single incoming event. In this kind of models, each atomic component receives one input per each external event. Consequently, the number of external and internal transitions to be executed is equal to the number of existing atomic components. Thus,

$$\textbf{\# Internal Transitions = \# Atomic Models}$$

$$\textbf{\# External Transitions = \# Atomic Models}$$

$$\begin{aligned}\textbf{Time spent per external event} = \; & \textbf{[(\# External Transitions * TimeInExternalTransition) +} \\ & \textbf{(\# Internal Transitions * TimeInInternalTransition)]}\end{aligned}$$

This information is essential to carry out the performance analysis.

2.2.2 Type-2 models

Model type 2 has more interconnections between the components of each coupled model. The inner atomic components are interconnected; therefore, there is a greater number of messages interchanged in these kinds of models and the overhead grows accordingly.

2.2.2.1 Sample Type-2 model

The following is a sample model with four levels of depth and a width of four components (as explained before, we have in this case four components per level, three of which are atomic and the remaining one is coupled).

The *Top model* (*Coupled Component #0*) is shown below.

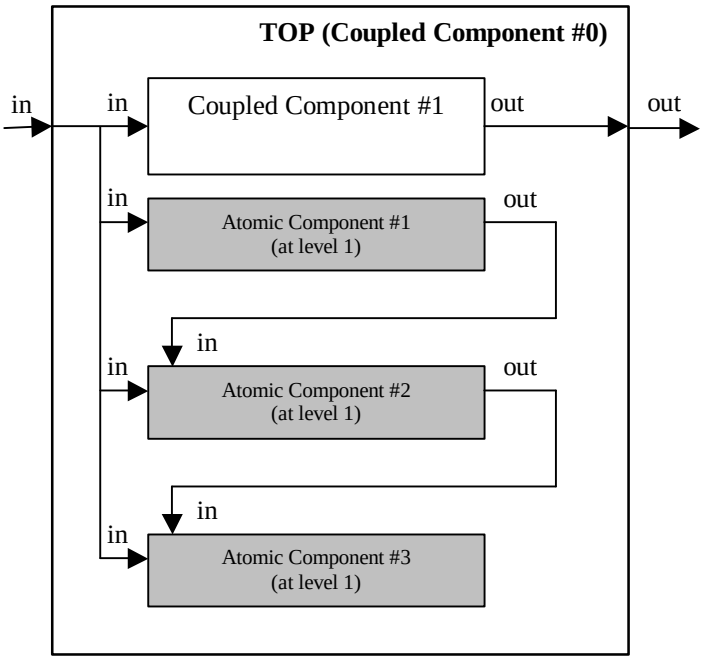


Figure 10: Top model (type 2)

The figure shows the greater number of connections in comparison with Type-1 models. *Coupled Component #1* is also formed by three atomic components and one coupled model (*Coupled Component #2)*. The same structure can be found in *Coupled Component #2* that is composed by *Coupled Component #3*. Neither *Coupled Component #1* nor *Coupled Component #2* will be shown here because of its similarity to those components shown before.

Finally, *Coupled Component* #3 is quite simple, because it is the last coupled component in the obtained hierarchy.

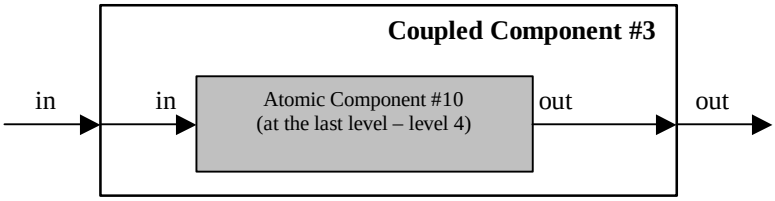


Figure 11: Coupled Component #3 (type 2)

2.2.2.2 Characteristics of Type-2 models

The number of both atomic and coupled components in type-2 models and those of type-1 models are equal. Then, given a specified *d* depth and *w* width, we obtain *d* coupled components with *w-1* atomic components inside each coupled component (except for the last coupled model which only includes one atomic component).

The total number of atomic components in a model generated by the tool is:

 # Atomic Models = (width – 1) * (depth – 1) + 1

However, the interconnections in Type-2 models are not the same as those in Type-1 models. Therefore, the number of internal and external transition also differs from the previous type.

When an external event is received, it is transmitted through the input port to the *top* model and to all its components. Similarly, the inner-coupled components retransmit the event accordingly.

Additionally, each time an atomic component sends an output, another atomic component receives it through its input port.

As we can see, the number of internal and external transitions executed upon the reception of an external event is much greater in Type-2 models. Thus,

$$\textbf{\# Internal Transitions} = \Sigma_{(i=1\,..\,w-1)}\ \textbf{i * (d – 1) + 1}$$

$$\textbf{\# External Transitions} = \Sigma_{(i=1\,..\,w-1)}\ \textbf{i * (d – 1) + 1}$$

In the above example,

$$\textit{\# Atomic Models} = (4 - 1) * (4 - 1) + 1 = \textbf{10}$$

and,

$$\textit{\# Internal Transitions} = \Sigma_{(i=1\,..\,4-1)}\ i * (4 - 1) + 1 = 18 + 1 = \textbf{19}$$

$$\textit{\# External Transitions} = \Sigma_{(i=1\,..\,4-1)}\ i * (4 - 1) + 1 = 18 + 1 = \textbf{19}$$

2.2.3 Type-3 models

Type-3 models are comparable to Type-2 models, but some differences exist. This new kind of model also connects the outputs of its inner atomic components to an auxiliary output, thus generating even more overhead in the simulation due to the message exchange.

2.2.3.1 Sample Type-3 model

The topmost component of a model with a depth of four and a width of four is depicted below.

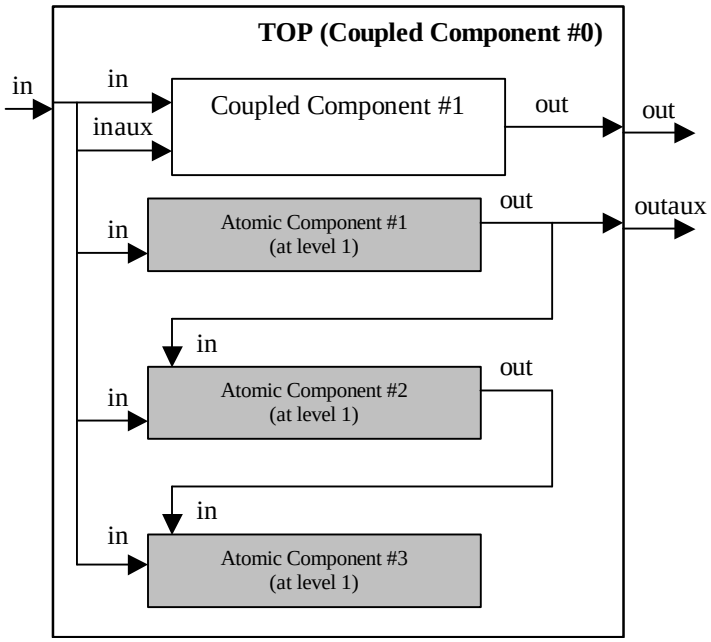


Figure 12: Top model (type 3)

The *Coupled Component #1* is very similar to the *top model*:

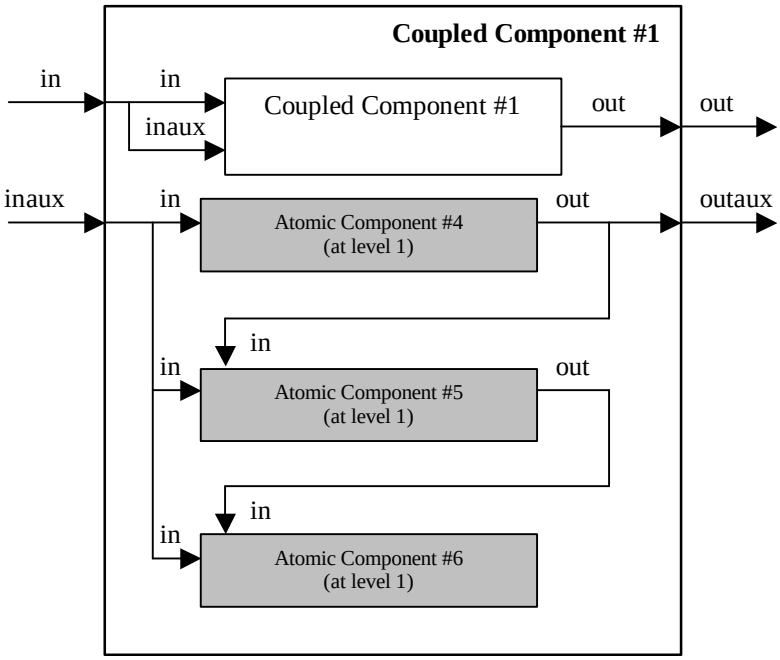


Figure 13: Coupled Component #1 (type 3)

Coupled Component #2 repeats the structure shown in the *Coupled Components #1*.

Finally, *Coupled Component #3* is a simple component and contains only one atomic component:

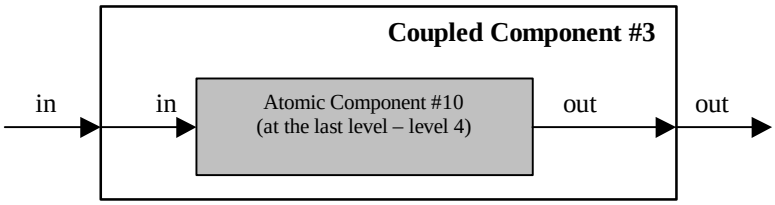


Figure 14: Coupled Component #3 (type 3)

2.2.3.2 Characteristics of Type-3 models

Type-3 models have even more interconnections than Type-2 models. In this case, not only auxiliary input ports but also auxiliary output ports generate more overhead in message exchange among components.

The difference between Type-2 and Type-3 models is the inclusion of these auxiliary ports. Then, the number of both coupled and atomic components per model remains the same as in Type-2:

$$\#\ Atomic\ Models = (width - 1) * (depth - 1) + 1$$

$$\#\ Internal\ Transitions = \Sigma_{(i=1\ ..\ w-1)}\ i * (d - 1) + 1$$

$$\#\ External\ Transitions = \Sigma_{(i=1\ ..\ w-1)}\ i * (d - 1) + 1$$

Hence, in the previous example,

$$\#\ Atomic\ Models = (4 - 1) * (4 - 1) + 1 = 10$$

$$\#\ Internal\ Transitions = \Sigma_{(i=1\ ..\ 4-1)}\ i * (4 - 1) + 1 = 18 + 1 = 19$$

$$\#\ External\ Transitions = \Sigma_{(i=1\ ..\ 4-1)}\ i * (4 - 1) + 1 = 18 + 1 = 19$$

2.3 Performance analysis

Thorough testing was developed to analyze the simulator performance under different conditions. The analysis compares the overhead obtained when using the following versions of the toolkit:

- Original stand-alone CD++ simulator
- Parallel CD++ simulator with *NoTime* (unsynchronized) kernel
- Parallel CD++ simulator with *TimeWarp* (optimistic) kernel

Recall that the parallel version supports not only parallel but also stand-alone simulation. All the testing developed in this work is carried out in stand-alone fashion. Results on parallel performance are analyzed in [Tro01b].

The layers involved in each technique in stand-alone execution are shown in the next figure. The MPI layer, which is stripped in the chart, is not used when stand-alone execution is performed with the parallel simulator.

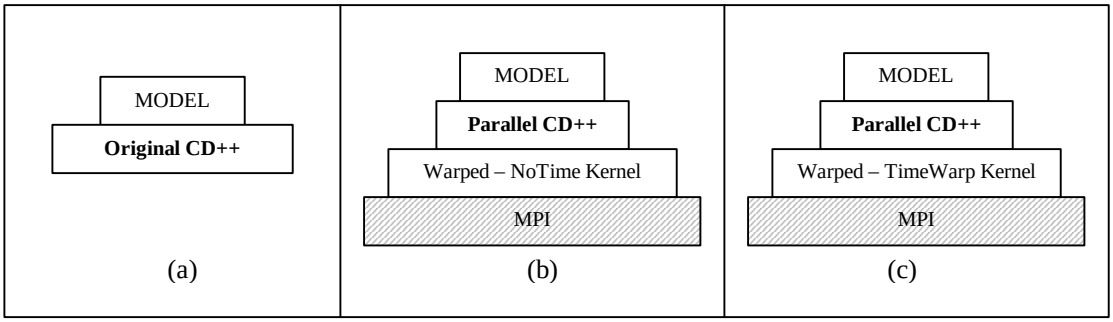


Figure 15: Comparison of the layers involved in (a) Original CD++, (b) Parallel CD++ with NoTime kernel, and (c) Parallel CD++ with TimeWarp optimistic kernel

Models with different shapes and sizes have been generated in order to simulate diverse model characteristics and workloads.

2.3.1 Test notes

The testing described in this chapter was performed at the **RADS Laboratory**, Systems and Computing Engineering Department, Carleton University (Ottawa, Canada). The testing was run on the *Alpha measurement network*, with Pentium computers with 128 MB of RAM.

The installed operating system was *Red Hat Linux 6.2*.

2.3.2 DEVS models

Different shapes and behaviors of DEVS models were tested. We used sample models created with the synthetic generator in order to exemplify some of the obtained results.

The *theoretical execution time* for a given simulation does not include any overhead. It is basically the sum of all time spent in internal and external transition all along this simulation.

Total theoretical time = [(# External Transitions * TimeInExternalTransition) + (# Internal Transitions * TimeInInternalTransition)] * NumberOfEvents

This value is shown in the charts and is compared with the obtained execution times for each simulation technique that includes the associated overhead.

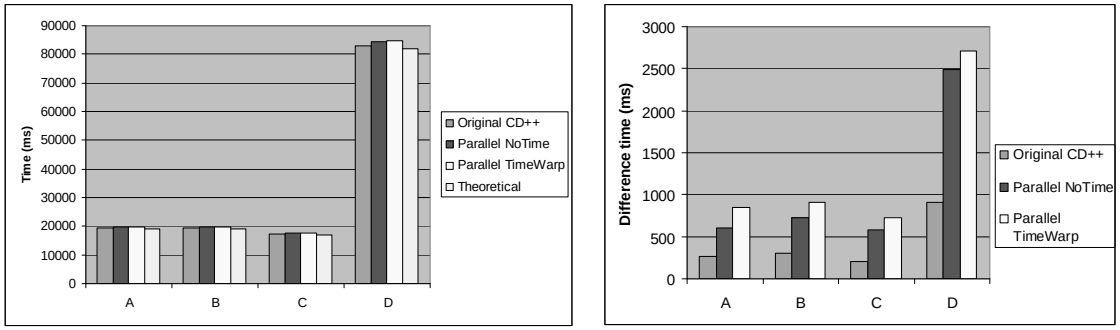
2.3.2.1 Type-1 models

Table 1 presents the parameters used in the simulations, which are labeled with capital letters, and their associated values.

Simulation	Model Type	Depth	Width	Internal Transition	External Transition
A	1	3	10	50 ms	50 ms
B	1	10	3	50 ms	50 ms
C	1	5	5	50 ms	50 ms
D	1	10	10	50 ms	50 ms

Table 1: Simulation parameters (Type-1 models)

The experiments have been performed with a workload of 10 external events per simulation. The next figure shows the execution times for each technique and a comparison of results.



(a) **(b)**

Figure 16: Performance results on Type-1 Models. (a) Execution times for Type-1 DEVS models, (b) Difference between experiments and theoretical time

Figure 16 (a) shows the different execution times for Type-1 models using the available simulation techniques. The theoretical time is included for comparison. *Figure 16 (b)* shows the difference between execution time and theoretical time.

The amount of overhead is measured by subtracting the theoretical time from the execution time and dividing that by the execution time itself, that is:

$$Overhead\ (\%) = \frac{(executionTime - theoreticalTime)}{executionTime}$$

The next figure presents the overhead incurred by each abstract simulator.

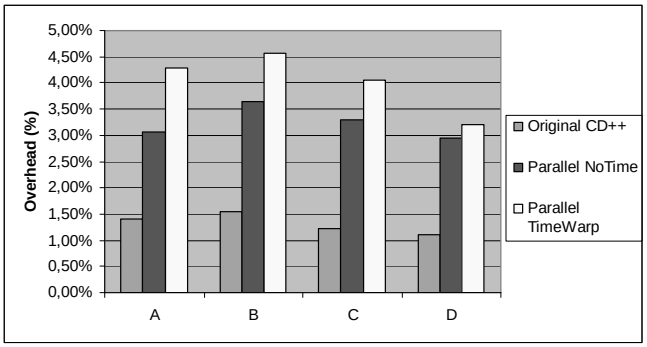


Figure 17: Overhead incurred by the abstract simulators for Type-1 models

As we can see, each technique induces a different overhead to the simulation. The charts show that the original CD++ technique is the one that executes with minimum overhead. When executing the parallel abstract simulator, the NoTime kernel adds less overhead than the TimeWarp kernel.

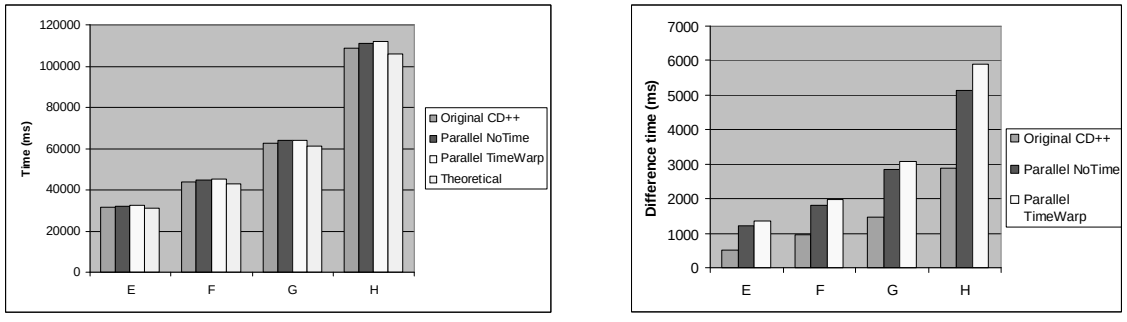
2.3.2.2 Type-2 models

Table 2 presents the parameters used in the simulations and their associated values for Type-2 models.

Simulation	Model Type	Depth	Width	Internal Transition	External Transition
E	2	3	6	50 ms	50 ms
F	2	6	3	50 ms	50 ms
G	2	5	5	50 ms	50 ms
H	2	6	6	50 ms	50 ms

Table 2: Simulation parameters (Type-2 models)

The experiments have been performed with a workload of 10 external events per simulation. The following figure shows the obtained results.



(a) **(b)**

Figure 18: (a) Execution times for Type-2 DEVS models, (b) Difference between experiments and theoretical time

Figure 18 (a) shows the different execution times for Type-2 models using the available simulation techniques, while *Figure 18 (b)* shows the difference between the execution time and theoretical time.

Figure 19 presents the overhead incurred by each abstract simulator for Type-2 models.

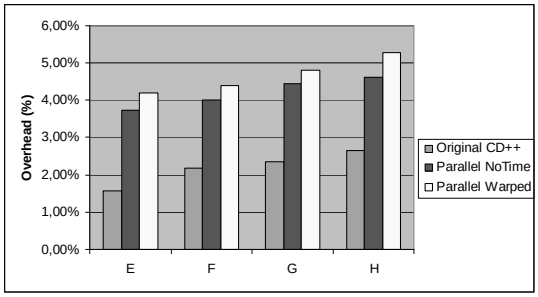


Figure 19: Overhead incurred by the abstract simulators for Type-2 models

Again, the chart shows that the original CD++ technique is the one that executes with minimum overhead. When executing the parallel abstract simulator, the NoTime kernel adds less overhead than the TimeWarp kernel.

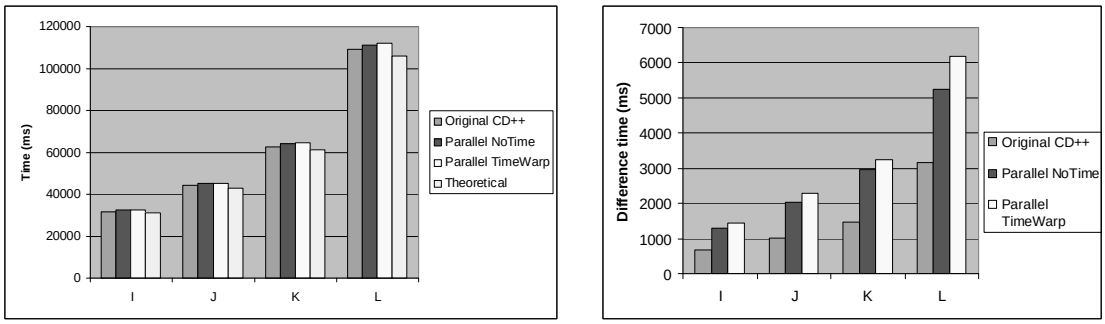
2.3.2.3 Type-3 models

Table 3 presents the parameters used in the simulations and their associated values for Type-3 models.

Simulation	Model Type	Depth	Width	Internal Transition	External Transition
I	3	3	6	100 ms	0 ms
J	3	6	3	0 ms	100 ms
K	3	5	5	50 ms	50 ms
L	3	6	6	50 ms	50 ms

Table 3: Simulation parameters (Type-3 models)

The experiments have been performed with a workload of 10 external events per simulation. The following figure shows the obtained results.



(a) **(b)**

Figure 20: (a) Execution times for Type-3 DEVS models, (b) Difference between experiments and theoretical time

Figure 20 (a) shows the different execution times for Type-3 models using the available simulation techniques, while *Figure 20 (b)* shows the difference between the execution time and theoretical time.

Lastly, *Figure 21* presents the stable overhead incurred by each abstract simulator for Type-3 models.

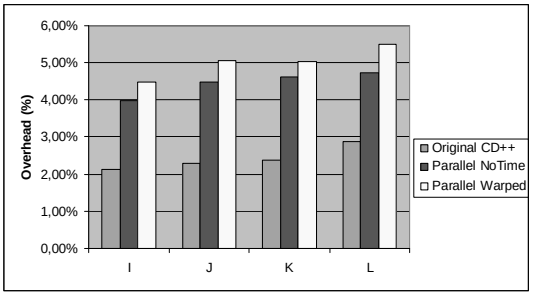


Figure 21: Overhead incurred by the abstract simulators for Type-3 models

2.3.2.4 Models without workload in transition functions

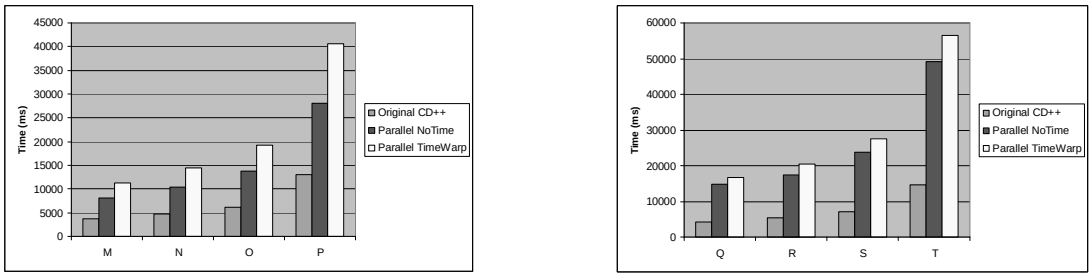
In the previous sections, test cases executed different ranges of workload in their internal and external transition functions. Time consuming Dhrystone code was used to resemble instructions to be executed in both atomic transition functions. This section studies a different type of model.

These test cases were also produced with our synthetic model generator. However, there is no Dhrystone code to be executed in the external and internal transition functions. Therefore, all the execution time corresponds to the overhead incurred by the different simulation techniques under study. The following table shows the different model parameters.

Simulation	Model Type	Depth	Width	Internal Transition	External Transition
M	1	5	10	0 ms	0 ms
N	1	10	5	0 ms	0 ms
O	1	8	8	0 ms	0 ms
P	1	10	10	0 ms	0 ms
Q	3	5	10	0 ms	0 ms
R	3	10	5	0 ms	0 ms
S	3	8	8	0 ms	0 ms
T	3	10	10	0 ms	0 ms

Table 4: Simulation parameters for models without workload

Experiments have been performed with a workload of 1000 external events per simulation. Again, the next figure shows the execution times for each technique in order to compare the obtained results.



(a) **(b)**

Figure 22: Execution times for pure-overhead simulation for (a) Type-1 models, (b) Type-3 models

In summary, these test cases show the execution time of simulations that do not execute any Dhrystone code at all in their internal and external transition functions. The presented results illustrate the time consumed on carrying on the simulation in presence of a considerable quantity of external events. Actually, since all the execution time is consumed by the simulator in all these cases, the resulting overhead in every case is 100%.

2.4 Conclusions about performance analysis

We have developed a synthetic benchmarking tool that can be applied to DEVS environments to analyze CD++ performance easily and thoroughly. Different types of models were tested automatically, showing that we can execute models paying a small cost in terms of processing overhead.

As we have shown, each simulation technique has an associated overhead that depends on the size, the shape and the behavior of the simulated model. We found that even with medium and large-scale models, the simulation can be carried out properly and the obtained overhead is of a manageable size and remains stable. The original CD++ tool executes with minimum overhead and therefore it is an appropriate tool when stand-alone execution is adequate. The NoTime kernel outperforms the TimeWarp kernel when using the parallel simulator.

3. REAL-TIME EXTENSION TO THE CD++ TOOLKIT

In the previous chapter, simulation techniques available in CD++ have been analyzed. It has been shown that a relatively small overhead is paid when executing mid to large-scale models. The performance analysis showed that a real-time extension to the tool was feasible.

3.1 Virtual time simulation approach

The existing techniques in the CD++ toolkit employ a *virtual time* approach. The methodology is useful for non-interactive simulation. This strategy advances the time disregarding any real clock attached to the simulation mechanism and periods of inactivity are skipped by the tool. In contrast to a *real time* simulation, it is useless to connect inputs and outputs to the environment when the *virtual time* simulation is performed, because the time in the simulation framework does not evolve at the same speed as within its surroundings.

In order to execute a simulation using the *virtual time* approach, CD++ maintains a variable in which the current *simulation time* is stored and updated. Again, note that this value is not linked at all to any physical clock. The update of that variable is performed by the simulator as follows. When the simulation starts, this *simulation time* is initialized to zero. Then, the imminent event (*i.e.* the event with the earliest time of occurrence) is computed and the *simulation time* is advanced accordingly in order to process that event. Once it has been processed completely, the new imminent event is computed, the *simulation time* is advanced and the new event is processed. This cycle of advancing the *simulation time* and processing the imminent event is repeated. The model execution ends when the *simulation time* reaches the *stop time* indicated by the user, or else when there are no more pending events.

The execution of a model using the *virtual time* approach with the CD++ toolkit is illustrated in the following example.

3.1.1 Sample model simulation using virtual time

Figure 23 shows a top-model that is formed by two coupled components, four input ports and four output ports. Each inner-coupled component is composed of two simple atomic models. An atomic component is linked to the environment through one input port and one output port. The workload that is executed in each component varies from one atomic component to another. The following figure shows the entire model structure.

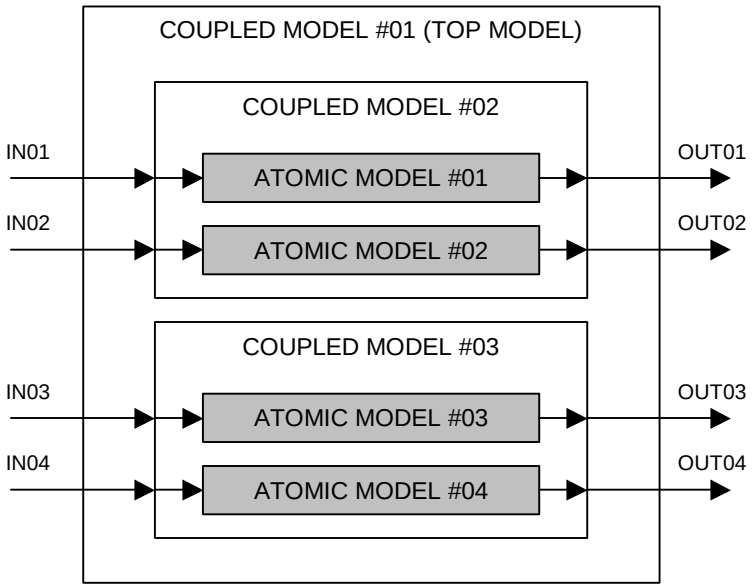


Figure 23: Sample model

External events can be received by the model through any of its input ports. This information is supplied by the user in the **event file**, where times are written in the *hours:minutes:seconds:milliseconds* format. An event file for this sample model is shown below.

event time	*input port*	*value*
00:00:05:000	in01	1
00:01:28:100	in02	1
00:18:21:000	in03	1
00:31:15:500	in04	1
00:45:30:200	in02	1
01:05:00:500	in01	1
02:15:00:900	in04	1
05:50:30:200	in03	1

Figure 24: Sample event file for the given model using the virtual time approach

CD++ executed the previous model with this event file. Once the simulation has ended, results can be found in an output file. The next figure shows the obtained **output file**, illustrating how time evolves when the virtual time approach is used in a simulation.

actual output time (physical or wall-clock time)	*output time (simulation-time)*	*output port*	*value*
00:00:00:060	00:00:05:000	out01	1
00:00:00:130	00:01:28:100	out02	1
00:00:00:230	00:18:21:000	out03	1
00:00:00:320	00:31:15:500	out04	1
00:00:00:390	00:45:30:200	out02	1
00:00:00:430	01:05:00:500	out01	1
00:00:00:520	02:15:00:900	out04	1
00:00:00:620	05:50:30:200	out03	1

Figure 25: Output file using the virtual time approach

The first column indicates the *physical time* (i.e. *wall-clock time*) at which the output has been sent. It is the time elapsed since the beginning of the simulation execution. The *simulation-time* associated to each message is shown in the second column. The *output port* and the *associated value* that has been sent are exhibited in the third and fourth columns respectively.

For instance, the first line describes an output sent through port *out01* with a value of *1*. That output has been performed at the *simulation time 00:00:05:000*, but the corresponding *physical time* at that moment was *00:00:00:060*.

Recall that, when the *virtual time* approach is used, all periods of inactivity are skipped, which leads to the particularity shown above when executing a model. In conclusion, we can see clearly that events are not processed at their actual scheduled times.

3.2 Real-time simulation approach

Modifications have been developed in order to allow real-time simulation in the CD++ toolkit. A real time system is defined as a system whose correctness depends not only on the logical results of computation, 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 handle events in a timeliness fashion where time constraints can be stated and validated. These new features would allow interaction between the simulator and the surrounding environment. Therefore, inputs could be received by ports connected to real input devices such as sensors, timers, thermometers or even data collected from human interaction. Similarly, outputs could be sent through output ports connected to devices such as motors, transducers, gears, valves or any other component.

3.2.1 Time advance in the simulation process

In order to implement the real-time extension to the toolkit, advance of the simulation-clock must be tied to the wall-clock (*i.e.* physical time). To do so, the *root coordinator* has been modified to provide this functionality.

The *root coordinator*, inheriting from the *coordinator* class in the processor hierarchy in CD++ (see Chapter 1 for further details), manages the time advance along the execution of a simulation. In addition, it is responsible of starting each new simulation cycle by issuing the corresponding message. When the *virtual time* approach was used, the messages were immediately generated and sent by the *root coordinator* to initiate the new cycle.

Nevertheless, when the *real-time* simulation is performed, the coordinator must wait until the physical time reaches the next event time to initiate the new cycle. A new simulation cycle can be started due to:

- the reception of an *external event*, or
- the consumption of time indicated by *ta(s)*

Evidently, periods of inactivity are not skipped in the real-time extension. The simulation process remains quiescent while these periods are being experienced. Instead of forcing a time advance up to the next programmed event and thus anticipating the execution of a programmed task, the *root coordinator* expects the scheduled time to be reached and only then starts the new simulation cycle. Hence, messages interchanged between processors are sent, ideally, at their actual scheduled time. However, this ideal timely processing of events may not be obtained if the incurred overhead degrades performance greatly.

3.2.2 Adding deadlines in the real time model execution

Timeliness along a simulation is a substantial property in the real time approach. When a model is being executed using this technique, it is usually 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.

Typically, a model has to react to an external event within a given time to produce an output in order to solve a given problem. For instance, in case of having a sensor indicating dangerous overheat, an energy plant needs to shut down a part of its system within given period of time.

A way to indicate a deadline time for an external event is provided in the real time extension of the toolkit. The new extended format of the **event file** is illustrated in the next figure.

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 26: Format of the event file in the real time extension

As we can see, not only an *associated deadline* but also an *output port* must be indicated in the new event file. Thus, 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.

3.2.3 Sample model simulations using the new real time approach

A real time simulation is exemplified using the model previously shown. The new event file is illustrated in the next figure.

event time	associated deadline	input port	associated output port	value
00:00:05:000	00:00:05:020	in01	out01	1
00:01:28:100	00:01:29:000	in02	out02	1
00:18:21:000	00:18:21:050	in03	out03	1
00:31:15:500	00:31:15:540	in04	out04	1
00:45:30:200	00:45:30:270	in02	out02	1
01:05:00:500	01:05:01:500	in01	out01	1
02:15:00:900	02:15:00:980	in04	out04	1
05:50:30:200	05:50:30:350	in03	out03	1

Figure 27: Sample event file for the given model using the real time approach

The file exhibits not only event times, but also their associated deadline information for each external event. For example, the result for the event arrived at time *00:00:05:000* through the input port *in01* must be sent before *00:00:05:020* through the output port *out01*. This states that the model must react to the given event in less than 20 milliseconds.

Now, the toolkit prints the *actual output time*, the *simulation-time* and the *associated deadline* for each event in the **output file**.

Additionally, in the *result* column one of these two values is obtained:

succeeded if **actual output time** ≤ **associated deadline**

not succeeded if **actual output time** > **associated deadline**

The following figure shows the corresponding output file for the executed model.

actual output time (physical or wall-clock time)	output time (simulation-time)	Associated deadline	result	output port	value
00:00:05:060	00:00:05:000	00:00:05:020	**not succeeded**	out01	1
00:01:28:070	00:01:28:100	00:01:29:000	**succeeded**	out02	1
00:18:21:090	00:18:21:000	00:18:21:050	**not succeeded**	out03	1
00:31:15:580	00:31:15:500	00:31:15:540	**not succeeded**	out04	1
00:45:30:270	00:45:30:200	00:45:30:270	**succeeded**	out02	1
01:05:00:560	01:05:00:500	01:05:01:500	**succeeded**	out01	1
02:15:00:980	02:15:00:900	02:15:00:980	**succeeded**	out04	1
05:50:30:290	05:50:30:200	05:50:30:350	**succeeded**	out03	1

Figure 28: Output file using the real time approach

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

time for the given event. It is possible to check whether the deadline has been met (*i.e.* the actual output time ≤ the associated deadline) looking at the fourth column. Finally, the *output port* and the obtained *value* are shown in the remaining columns.

For instance, the first line in the output file shows a deadline that has not been met in the execution. The associated time constraint was set at *00:00:05:020*, while the actual output time was *00:00:05:060*. Consequently, *not succeeded* is printed in that line, along with the output port *out01* and the value that has been transmitted. On the other hand, the second line shows an event whose deadline that has been successfully met. It was sent by the simulator at *00:01:28:070* while its associated deadline was *00:01:29:000*. In this particular case, five out of eight events have been processed on time.

3.2.3.1 Alarm-clock sample model

Different types of models can be executed with the new real-time extension CD++. An alarm clock model [Jac01] has been used to analyze the real-time constraints under the new simulation approach. This model can be thought of as a part of a more complex system. The model, which has an important component of time, is presented here.

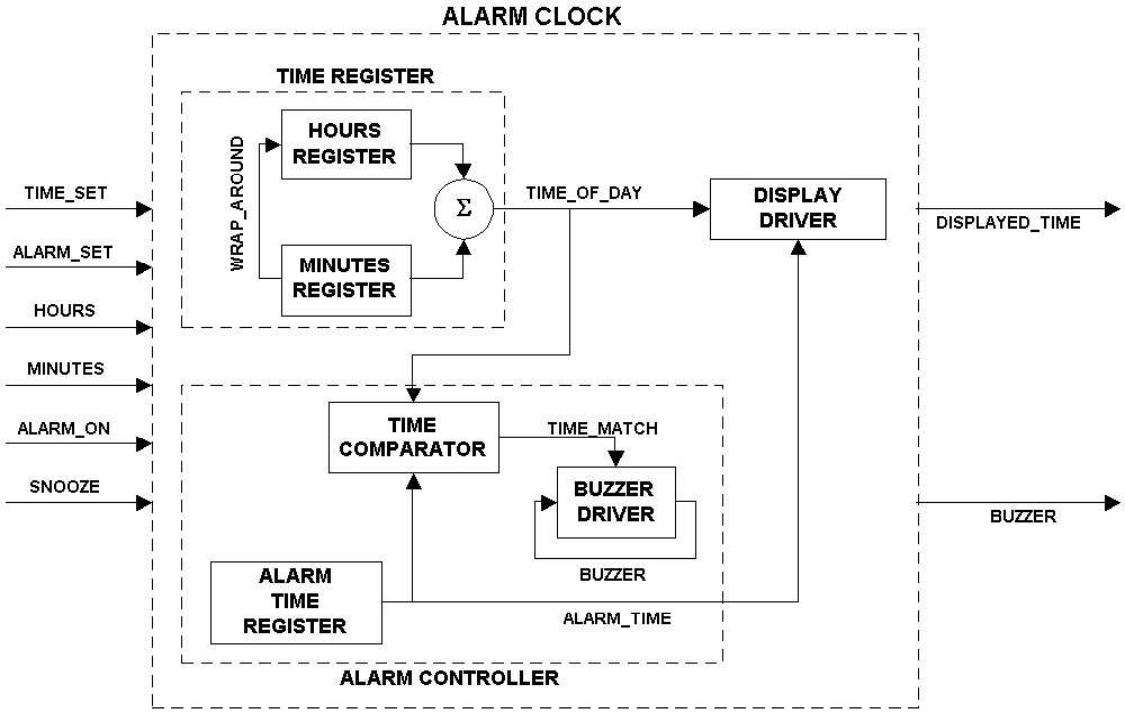


Figure 29: Alarm-clock conceptual model [Jac01]

The entire model has three levels in the hierarchy. The top level is the *ALARM CLOCK*. It has six input signals representing the push buttons and switch positions that exist in the real system. The input port *TIME_SET* is used in combination with *HOURS* and *MINUTES* to set the time of day. Similarly, the input port *ALARM_SET* is used in conjunction with *HOURS* and *MINUTES* to set the desired alarm time. The buzzer sounds if *ALARM_ON* is set and the actual time (*i.e.* time of day) is equal to the alarm time. *SNOOZE* stops the buzzer for a period of 10 minutes after which the buzzer will automatically sound again if *ALARM_ON* is set. The model has two output ports: *DISPLAY_TIME* represents the four-digit display, while *BUZZER_ON* represents the output of the buzzer speaker.

The top model can be subsequently decomposed into sublevels. The first sublevel consists of three components: the *TIME_REGISTER* which holds and automatically increments the time of day, the *ALARM_CONTROLLER* which holds the alarm time and decides whether the buzzer should be turned on or off. The third component is an atomic component named *DISPLAY_DRIVER*, which determines if *time of day* or *alarm time* must be displayed.

The second sublevel consists of five different atomic components. The *HOURS_REGISTER* and *MINUTES_REGISTER* respectively hold the hours and minutes that make up the time of day. The *TIME_COMPARATOR* compares the current time with the alarm time to detect a match and potentially sound the buzzer. The *ALARM_TIME_REGISTER* holds the alarm time. Finally, the *BUZZER_DRIVER* decides when the buzzer needs to be activated or deactivated.

The following is an excerpt from the output file produced by the simulation of the alarm clock.

actual output time (physical or wall-clock time)	***output time (simulation-time)***	***output port***	***value***
00:01:00:000	00:01:00:000	DISPLAY_TIME	00:01
00:02:00:000	00:02:00:000	DISPLAY_TIME	00:02
00:03:00:000	00:03:00:000	DISPLAY_TIME	00:03
(…)	(…)	(…)	(…)
00:30:00:000	00:30:00:000	DISPLAY_TIME	00:30
00:30:00:000	00:30:00:000	BUZZER_ON	1
00:31:00:000	00:31:00:000	DISPLAY_TIME	00:31
00:32:00:000	00:32:00:000	DISPLAY_TIME	00:32
00:32:45:500	00:32:45:500	BUZZER_ON	0
00:33:00:000	00:33:00:000	DISPLAY_TIME	00:33
00:34:00:000	00:34:00:000	DISPLAY_TIME	00:34
00:35:00:000	00:35:00:000	DISPLAY_TIME	00:35
00:36:00:000	00:36:00:000	DISPLAY_TIME	00:36
00:37:00:000	00:37:00:000	DISPLAY_TIME	00:37
00:38:00:000	00:38:00:000	DISPLAY_TIME	00:38
00:39:00:000	00:39:00:000	DISPLAY_TIME	00:39
00:40:00:000	00:40:00:000	DISPLAY_TIME	00:40
00:41:00:000	00:41:00:000	DISPLAY_TIME	00:41
00:42:00:000	00:42:00:000	DISPLAY_TIME	00:42
00:42:45:500	00:42:45:500	BUZZER_ON	1
00:43:00:000	00:43:00:000	DISPLAY_TIME	00:43
(…)	(…)	(…)	(…)

Figure 30: Output file excerpt - Execution of the Alarm-clock [Jac01] in real-time

Figure 30 shows results obtained after the execution of the alarm clock using the real-time approach. Generally, we can see that as time passes, the actual time is obtained through the *DISPLAY_TIME* port which resembles the usual digital display of an alarm clock. In addition, information about the buzzer alarm is obtained in the output file.

The buzzer is turned on at *00:30:00:000* and this is notified through the output port *BUZZER_ON* at that time. The time still evolves normally and the actual time is obtained through the *DISPLAY_TIME* port. The user turns off the buzzer at *00:32:45:500*, where the *BUZZER_ON* issues a *0*. Recall that the buzzer can be deactivated

using the *SNOOZE* button, but the alarm will buzz again after an idle period of ten minutes. Hence, at time *00:42:45:500* the buzzer is turned on again, when the output port *BUZZER_ON* issues a *1*.

It is important to point out that actual output-times are equal to their corresponding simulation times. This fact shows that delays are remarkably small all along the simulation of this alarm clock. Therefore, such simulation could meet easily the deadlines imposed by the user.

3.2.3.2 Vending machine sample model

Moreover, a vending machine model [Li01] has been used for further analysis of the real-time extension in CD++.

The simulated vending machine is similar to the ones that exist in some cafeterias. Different items can be purchased by inserting the sufficient amount of money and then selecting the appropriate button to dispense the desired product. The machine returns the correct amount of change, keeps track of how many items have been dispensed and informs out-of-stock products to the customer.

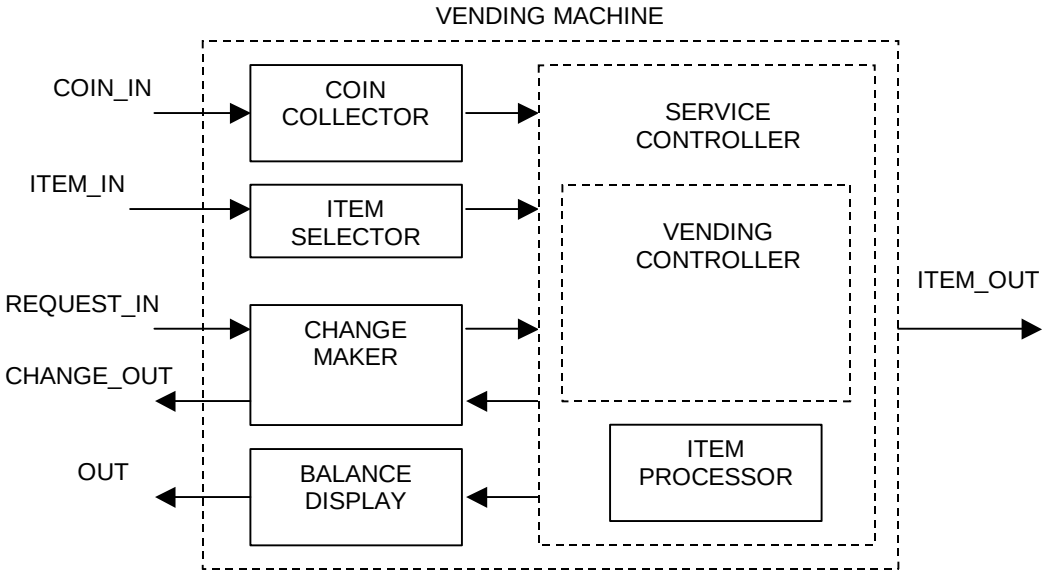


Figure 31: Vending machine conceptual model [Li01]

The system is composed of several atomic components (a *coin collector*, an *item selector*, a *change maker*, a *balance display*, an *item processor* and others) and coupled components (a *service controller* and a *vending controller* inside of it).

The model has three input ports. Coins are inserted through the *COIN_IN* port, items are selected through the *ITEM_IN* port and change is requested through the *REQUEST_IN* port. The output ports are used as follows: *ITEM_OUT* is used to dispense the products, *OUT* resembles the balance display of the machine and *CHANGE_OUT* is used for the returned coins.

The following figure shows a sample event file for the vending machine model. Here, a customer inserts different amounts of money and requests a particular item. Deadlines are imposed to each incoming event.

event time	associated deadline	input port	associated output port	value
00:00:10:000	00:00:12:500	COIN_IN	OUT	0.25
00:00:15:000	00:00:17:500	COIN_IN	OUT	1.00
00:00:20:000	00:00:25:500	COIN_IN	OUT	0.25
00:00:25:000	00:00:30:000	ITEM_IN	ITEM_OUT	28
(…)	(…)	(…)	(…)	(…)

Figure 32: External event file - Vending machine [Li01]

For instance, the first quarter is received through the *COIN_IN* port at time *00: 00:10:000*, and the associated output is expected through the port *OUT* before *00:00:10:250*. Then, a dollar (*1.00*) is received at time *00:00:15:000*, and so on. Finally, the item 28 is selected at time *00:00:25:000*.

actual output time (physical or wall-clock time)	output time (simulation-time)	output port	value
00:00:12:010	00:00:12:000	OUT	0.25
00:00:17:010	00:00:17:000	OUT	1.25
00:00:22:010	00:00:22:000	OUT	1.50
00:00:28:020	00:00:28:000	ITEM_OUT	28
00:00:30:010	00:00:30:000	OUT	0.00
(…)	(…)	(…)	(…)

Figure 33: Output file excerpt - Execution of the vending machine in real-time [Li01]

The previous figure shows the corresponding output file. The balance display is updated through the *OUT* port, two seconds after each coin is inserted. The item *28* is dispensed through the *ITEM_OUT* port at time *00:28:000*. Events are processed on time, and small differences can be observed between the *message time* and the *actual time* (*i.e.* wall-clock time) at which they have been produced.

3.3 Conclusions about the real-time extension in CD++

We have provided a means to execute models in real-time using the CD++ toolkit. A real-time simulation differs in different aspects from the existing techniques that used a virtual-time approach. A comparison between *virtual time* and *real time* approaches is summarized in the following table.

Virtual time approach	*Real time approach*
Previously available in the CD++ toolkit	Implemented in this work
Periods of inactivity are skipped	During periods of inactivity the simulator remains quiescent
Events are not processed based on physical time but only on simulated time	Simulation is tied to the physical clock, so events can be processed at their indicated scheduled time
Forbids a link between simulator and environment	A link between simulator and environment is feasible
Timing constraints do not exist	Deadlines can be associated to events. Timeliness can be tested and analyzed.

Table 5: Comparison between both simulation approaches

As we have shown, timeliness is an essential and meaningful characteristic of real time simulations. In such cases, whether a given deadline is met depends on several factors:

- *Overhead of the tool:* the execution of the simulation mechanism affects the overall performance. Usually, this overhead becomes larger as the size of the model increases, mainly because the time spent by exchanging messages among processors.

- *Workload in atomic components:* the more workload that has to be executed in internal and external transition functions, the more time that is needed to complete the execution of the corresponding code

- *Associated deadlines:* if the associated deadline for a given event is very tight, then it is not likely to be met. On the contrary, a loosened (relaxed) deadline is likely to be met more easily in a simulation.

The first factor, *overhead of the tool*, is intrinsically involved with the simulation process. It has to be minimized to allow a wide range of models to be executed properly using the real time simulation toolkit. The second factor, *workload in atomic components*, varies from one model to another and depends on the characteristics of the models under execution. Finally, the *associated deadline* influences the success of meeting a given deadline, and they are imposed by the user.

The next chapter provides a performance analysis of the CD++ real time simulator.

4. PERFORMANCE ANALYSIS OF THE REAL TIME SIMULATOR

This chapter analyzes the real time performance of the CD++ toolkit. A thorough testing process is carried out using different kinds of models.

4.1 Introduction

Recall from the previous chapters that a real time simulator must be able to interact with a surrounding environment. Timing correctness requirements in a real time system arise because of the physical impact of the controlling systems' activities upon its environment [Sta88]. Therefore, it is imperative to ensure a timely processing of events in our simulation. A thorough testing must be carried out on the real time simulator in order to understand its limitations and weaknesses. This section studies the results of several simulations using the real time extension of the CD++ toolkit.

For each real time simulation, essential data about the execution is stored for further analysis. This information includes:

- ***Number of missed deadlines:*** represents the number of deadlines that have been missed along the entire execution of a model. A deadline is **missed** if its *response time* is greater than its *associated deadline*.

- ***Worst-case response time:*** represents the maximum time between the arrival of an event and the output that the model produces in response, in the entire simulation process.

A wide set of models are tested in order to define accurately the performance of the real time simulator under different scenarios.

4.2 Test parameters

Different parameters are taken into account to analyze a given test case. These parameters are:

- ***Model size:*** it can be subsequently divided in *number of components per level* and *number of levels in the model hierarchy*.

- ***Number of interconnections between components:*** this parameter describes the complexity and characteristics of the existing interconnections in the model. This information is obtained by the model type when the synthetic generator is being used.

- ***Workload in transition functions:*** the number of milliseconds that have to be spent in the internal and external transition functions.

- ***Number of external events:*** the number of external events that are received along the entire simulation.

- ***Inter-event period:*** the period between an event and the following one. It describes the frequency of event arrival.

- ***Associated deadline:*** the deadline that has been associated to each incoming event. For instance, a deadline of 50 milliseconds means that the output for an event has to be issued within 50 milliseconds after the event arrival.

The first three parameters are intrinsically related to the model itself. They correspond to the specific characteristics of each model.

On the other hand, the last three parameters are involved with the simulation scenario under which the model is being executed. They are not related to the model, but to the constraints imposed by the user (*i.e.* the *associated deadline*) and the environment (*i.e.* the *number of external events* and *inter-event period*).

In the testing process, a wide set of parameters are used to analyze several cases of interest.

4.3 Test notes

The testing described in this chapter was performed in the **ParDEVS Laboratory**, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. All simulations were run on a *Compaq ProLiant 1600* node, consisting of a Pentium II 450MHz processor with 512 MB of RAM, 512-KB second-level ECC cache and 100-MHz GTL Bus.

The installed operating system was *Caldera OpenLinux*.

4.4 Test cases

Models are created with the synthetic generator whose implementation was described in Chapter 2. Tests results show both the *percentage of success* and the *worst-case response time* for each case. The former is obtained as follows,

$$\textit{Percentage of success} = \frac{(\textit{number of events} - \textit{number of missed deadlines}) * 100}{\textit{number of events}}$$

On the other hand, the ***worst-case response*** *time* is obtained as follows.

$$\textit{Worst-case response time} = \max(r_1, r_2, \ldots, r_N)$$

where r_i is the response time for the *i-th* event, and N is the *number of events* for the given simulation.

The experiments have been grouped in different categories that are described in the following subsections.

4.4.1 Varying number of levels in the hierarchy

4.4.1.1 Models without workload in the transition functions

Here, models have a fixed number of components per level, but the number of levels in the hierarchy (*i.e.* the *depth* that the model has) varies. In first place, Type-1 models are employed. These models have a small number of interconnections between their components.

Each execution receives 100 incoming events and a fixed inter-event period of 30 milliseconds. Neither the external transition function nor the internal transition function executes workload (time consuming code). Deadlines are imposed at 60 milliseconds after the event arrival. Actually, all events would be processed on time if the simulator did not add any overhead to the execution.

The following is an excerpt from the event file used in these experiments. It has 100 lines in total, one line for each incoming event.

event time	associated deadline	input port	associated output port	Value
00:00:05:000	00:00:05:060	in	out	1
00:00:05:030	00:00:05:090	in	out	1
00:00:05:060	00:00:05:120	in	out	1
00:00:05:090	00:00:05:150	in	out	1
(…)	(…)	(…)	(…)	(…)
00:00:34:940	00:00:35:000	in	out	1
00:00:34:970	00:00:35:030	in	out	1

Figure 34: Sample event file for the given model using the real time approach

For instance, the first event arrives through the *in* port at time *00:00:05:000* and its output must be issued though the *out* port before *00:00:05:060*. The second event arrives 30 milliseconds later, at *00:00:05:030*, whereas its associated deadline is *00:00:05:090*, and so on.

The following table summarizes all the information corresponding to the first test.

Simulation parameter	Associated value
Number of components per level	5 components
Number of levels in the hierarchy (Depth)	4 to 12 levels
Model type	Type-1
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	30 ms
Associated deadlines	60 ms
Number of atomic components in the obtained models	13 to 45
Number of coupled components in the obtained models	3 to 11

Table 6: Simulation parameters – Varying depth (number of levels in the hierarchy), Type-1 models

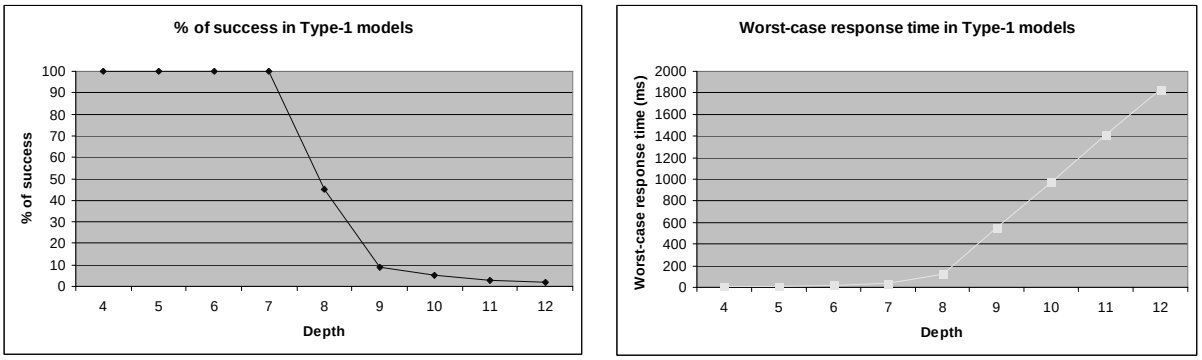
The total number of atomic and coupled components is also included in the table to provide more information about the executed models. These values are computed using the information about the synthetic model generator provided in Chapter 2. In this particular case, because of the parameters that have been used and specifically due to the varying depth, we have obtained models with a range of 13 to 45 atomic components and 3 to 11 coupled ones. Deeper models have not only more atomic components but also more coupled components in their structures.

Once the execution is over, we take into account the *number of deadlines* that have been met (*i.e.* the events that have been entirely processed before their associated deadlines) and the *number of missed deadlines* in order to measure the *percentage of success* for each simulation, as it was explained before.

Moreover, recall that CD++ stores the *worst-case response time* to enable a more comprehensive study of the real time performance. This value is also shown in the charts.

Notice that if the *worst-case response time* is smaller or equal than the *associated deadline* for a given simulation, then the *number of missed deadlines* is zero. This means that all events have been completely processed before their associated deadlines, and therefore we achieve a success of one hundred percent.

The following figure shows the corresponding charts for some Type-1 models.



(a) *(b)*

Figure 35: Real time execution of Type-1 models with varying depth
(a) Percentage of success, (b) Worst-case response time

Figure 35 (a) shows the percentage of success on Type-1 models when depth is variable. In these cases, the number of levels ranges from 4 to 12. Generally, deeper models have worse response times due to their larger size and increased complexity. Specifically, a noticeable number of deadlines are lost when the depth is eight or more in this particular case. The same phenomenon can be observed when we analyze the worst-case response times in these models with variable depth.

The second set of experiments shows a similar case for Type-3 models. Chapter 2 has shown that these models, with more interconnections between their components, are much more complex. Consequently, both the overhead and the number of executed transition functions are greatly increased. Because of this, we have relaxed the frequency of the incoming events. Now, the inter-event period is 40 milliseconds. The associated deadlines remain unchanged from the previous experiment.

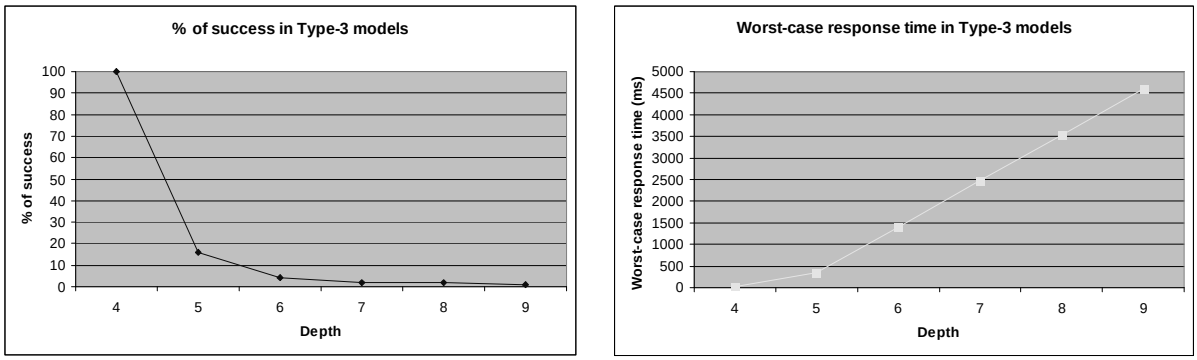
The following table shows the associated parameters.

Simulation parameter	Associated value
Number of components per level	5 components
Number of levels in the hierarchy	4 to 9 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	40 ms
Associated deadlines	60 ms
Number of atomic components in the obtained models	13 to 33
Number of coupled components in the obtained models	3 to 8

Table 7: Simulation parameters – Varying depth (levels in the hierarchy), Type-3

Now we have obtained models with a range of 13 to 33 atomic components and 3 to 8 coupled ones.

The following charts show the obtained results for these models.



(a) *(b)*

Figure 36: Real time execution of Type-3 models with varying depth
(a) Percentage of success, (b) Worst-case response time

The results are similar to those of Type-1 models. However, the results are not as successful as before. In Type-3 experiments, deeper models show a remarkable increase on response times, and therefore a low percentage of success is achieved even with a lower frequency of event arrival. The same fact will be observed in other experiments further in this work.

4.4.1.2 Models with workload in the transition functions

As a final study of models with varying depth, a different series of tests are presented here. In these models, time-consuming code is executed in their transition functions.

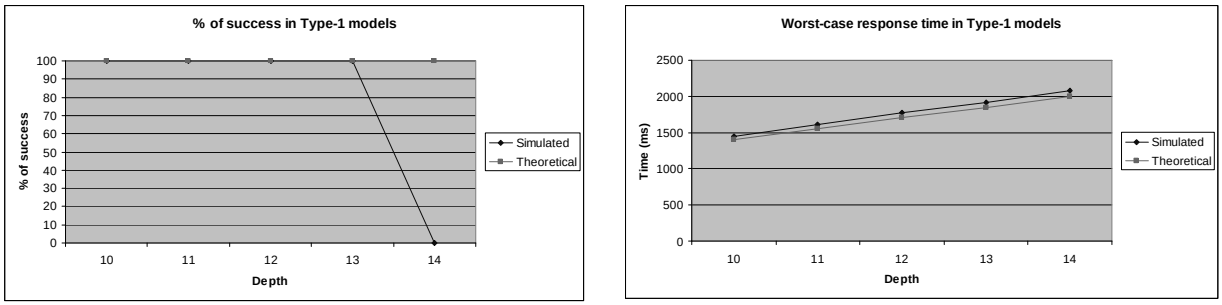
The following Type-1 models execute *50 milliseconds* of workload in the internal and external transition of their atomic components. The time-consuming Dhrystone code [Wei84] executed in the functions resembles the code that would be executed in a simulated model.

The table shows all the parameters that have been used.

Simulation parameter	Associated value
Number of components per level	4 components
Number of levels in the hierarchy	10 to 14 levels
Model type	Type-1
Workload in internal transition function	50 ms
Workload in external transition function	50 ms
Number of external events	100 events
Inter-event period	5000 ms
Associated deadlines	2000 ms
Number of atomic components in the obtained models	28 to 40
Number of coupled components in the obtained models	9 to 13

Table 8: Simulation parameters – Varying depth (levels in the hierarchy) with workload, Type-1

The following figures show the obtained results after the execution of these simulations. Note that, in addition, both the theoretical *percentage of success* and theoretical *worst-case response time* are shown in the charts. The theoretical results are simply the sum of all the time spent in executing the workload that is found in the internal and external transition functions. Neither the overhead incurred by the simulator nor any other factors that may affect simulation performance are included in the **theoretical** results. The series with the label **simulated** correspond to the real-time execution of these models, and therefore include all the overhead incurred in the simulations.



(a) *(b)*

Figure 37: Real time execution of Type-1 models with varying width and workload (a) Percentage of success, (b) Worst-case response time

Figure 37 (a) shows the *percentage of success* for Type-1 models with varying depth. The associated deadlines are imposed at 2000 milliseconds after the arrival of the event. A success of 100% is achieved in all cases, with the exception of one last case, which has 14 levels in the hierarchy and does not meet any deadline.

Figure 37 (b) shows the *worst-case response time* for this type of models with time-consuming code in the transition functions. These results are much more meaningful than those described in the previous sections because they allow a comparison between the simulated models and the theoretical cases. If we compare these experiments in which workload is executed in the internal and external transition, we can see that the incurred overhead is bounded. It remains nearly stable even if the depth is increased. The difference between the *theoretical worst-case response time* and the *simulated worst-case response time* is quite small.

4.4.2 Varying number of components per level

4.4.2.1 Models without workload in the transition functions

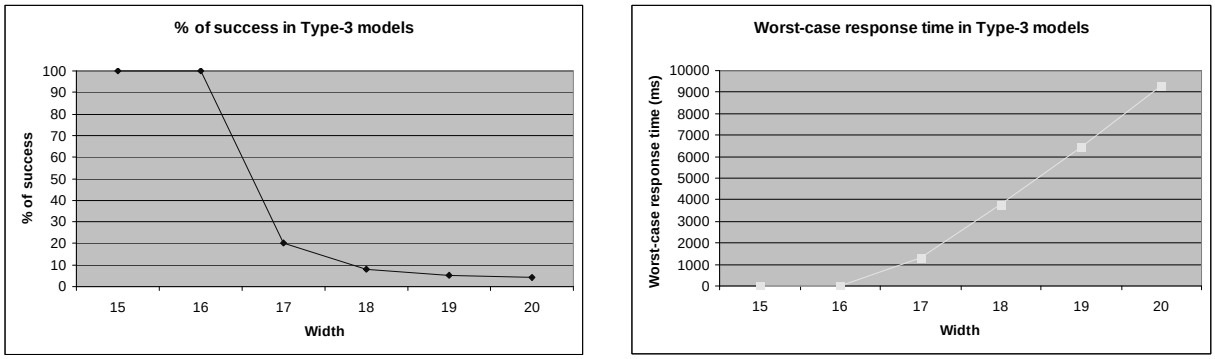
The previous subsection showed different models whose depth was variable and width was fixed. The following cases show models in which the depth is fixed, but the width (*i.e.* number of components per level) varies. These models do not execute workload in their transition functions. Larger models are used in these experiments. All parameters are described in the next table.

Simulation parameter	Associated value
Number of components per level	15 to 20 components
Number of levels in the hierarchy	3 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	200 ms
Associated deadlines	340 ms
Number of atomic components in the obtained models	29 to 39
Number of coupled components in the obtained models	2

Table 9: Simulation parameters – Varying width (components per level), Type-3

Now we have obtained models with a range of 29 to 39 atomic components. Due to the fixed depth, the number of coupled components is constant. Wider models, with more components per level, have more atomic components in their structures.

The following figure shows the obtained results for these models.



(a) *(b)*

Figure 38: Real time execution of Type-3 models with varying width (a) Percentage of success, (b) Worst-case response time

Figure 38 illustrates how component width can affect performance in a simulation. In this case, deadlines are more likely to be missed when the number of components per level is 17 or more.

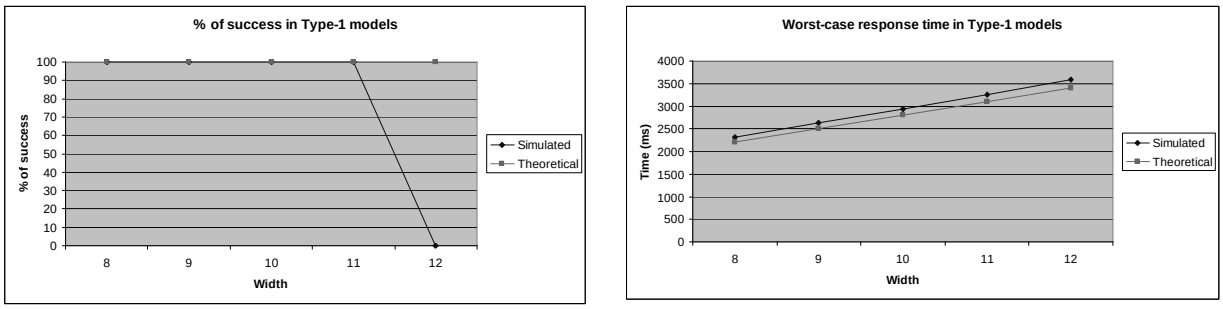
4.4.2.2 Models with workload in the transition functions

As we have shown before, it is interesting to study the execution of models whose internal and external transition functions execute time-consuming code. The following table describes the series of tests executed here.

Simulation parameter	Associated value
Number of components per level	8 to 12 components
Number of levels in the hierarchy	4 levels
Model type	Type-1
Workload in internal transition function	100 ms
Workload in external transition function	100 ms
Number of external events	100 events
Inter-event period	10000 ms
Associated deadlines	3400 ms
Number of atomic components in the obtained models	22 to 34
Number of coupled components in the obtained models	3

Table 10: Simulation parameters – Varying width (components per level) with workload, Type-1

Again, the **theoretical** series is also included in the charts. It shows the ideal results, where overhead is not taken into account. In contrast, the **simulated** series shows the obtained results using the toolkit.



(a) *(b)*

Figure 39: Real time execution of Type-1 models with varying width and workload (a) Percentage of success, (b) Worst-case response time

Figure 39 (a) illustrates the *percentage of success* for models with varying width and workload executed in the transition functions. The results are very similar as those described before when the depth was variable.

Figure 39 (b) shows the *worst-case response time* for this type of models. Again, these results are much more meaningful than those described in experiments that do not execute any workload. They allow the comparison between the simulated models and the theoretical cases. Moreover, the charts show that the incurred overhead is quite small, remaining nearly stable even for very wide models with more than 30 atomic components in their structures.

4.4.3 Varying number of components in the structure

The previous experiments have shown simulations where depth and width were variable. The analysis of a given case was independent from the others.

Here, we analyze the simulation of models taking into account the number of components in their structures. These experiments show the execution of different series of models:

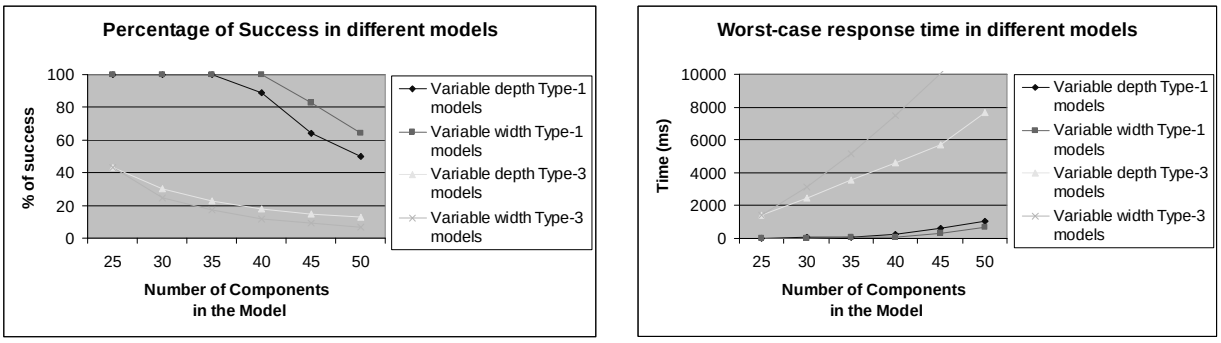
- *Type-1* models with varying *depth*
- *Type-1* models with varying *width*
- *Type-3* models with varying *depth*
- *Type-3* models with varying *width*

The goal is to compare the execution results for each series, examining the obtained results. The parameters are shown in the next table.

Simulation parameter	Associated value
Model type	Type-1 and Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	20 ms
Associated deadlines	1000 ms
Number of atomic components in the obtained models	25 to 50
Number of coupled components in the obtained models	5 to 10

Table 11: Simulation parameters – Comparison of models with varying depth and width

The results can be observed in the next charts. The number of components is variable because for each model, either the depth or the width is variable.



(a) *(b)*

Figure 40: Real time comparison of models with varying depth and width (a) Percentage of success, (b) Worst-case response time

Figure 40 (a) shows the *percentage of success* for Type-1 and Type-3 models when depth is variable and the width is fixed, and also when the width is variable and the depth is fixed. *Figure 40 (b)* illustrates the *worst-case response time* for each case.

Generally, the previous charts illustrate that if the number of components is increased, deadlines are more likely to be missed and therefore the *percentage of success* is reduced. This is because the number messages needed to perform the simulation grows in relation to the size and complexity of the model.

Particularly, the figures show that it is harder to simulate Type-3 models when the number of components increases due to their complex structure, in comparison with the equivalent (and more simple) Type-1 models. Consequently, the *worst-case response times* are remarkably increased for Type-3 models.

Type-3 models have a larger number of interconnections, and, consequently, the overhead introduced makes it harder to complete the entire simulation cycle on time.

Under these conditions, when a Type-1 model has 40 active components in its structure, more than 90 percent of success can be achieved. Alternatively, less than 20 percent of the deadlines are met for Type-3 models with 40 components in their structures.

4.4.4 Varying inter-event periods and associated deadlines

All previous cases have studied variations to the models themselves. A different approach is analyzed here, where the shape and behavior of the models remain unchanged but the scenario in which they are executed is modified. Different inter-event periods (*i.e.* the frequency of event arrivals) are employed. Consequently, we simulate external events that arrive at a different pace to the model, and analyze the behavior of the simulator under such circumstances. Furthermore, the impact of varying the associated deadlines is also tested.

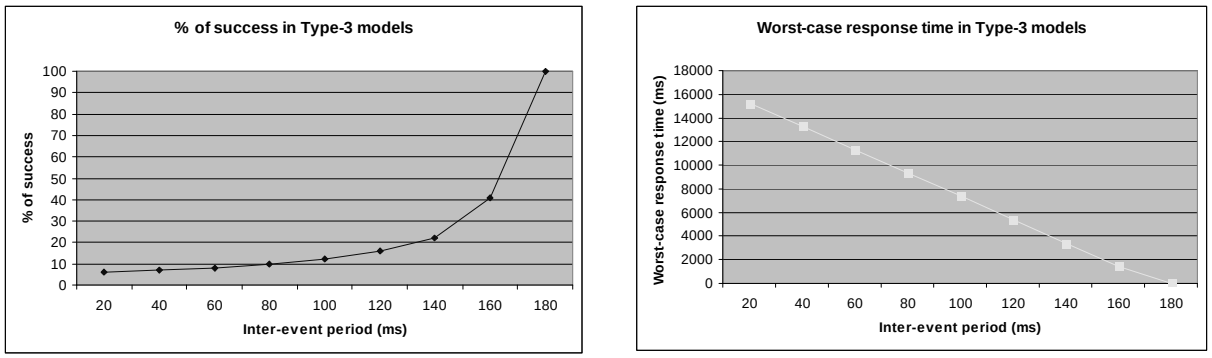
4.4.4.1 Varying inter-event periods

Events can arrive at a different pace, depending on the surrounding environment. In the first experiment, the inter-event periods varies from 20 to 180 milliseconds. Consequently, we can simulate events arriving at different pace along a simulation. Each simulation receives 100 external events from the environment. These experiments do not execute workload in the atomic transition functions. Parameters are shown in the next table.

Simulation parameter	Associated value
Number of components per level	5 components
Number of levels in the hierarchy	5 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	20 to 180 ms
Associated deadlines	1000 ms
Number of atomic components in the obtained models	50
Number of coupled components in the obtained models	7

Table 12: Simulation parameters – Varying inter-event period , Type-3

The following figure illustrates the obtained results for these models.



(a) *(b)*

Figure 41: Real time execution of Type-3 models with varying inter-event periods (a) Percentage of success, (b) Worst-case response time

Figure 41 shows that larger inter-event periods result on greater percentages of success. When the intervals between events become greater than 180 milliseconds, the simulator meets all the associated deadlines for the execution.

On the other hand, when the frequency of event arrival is extremely high, the *worst-case response times* become much longer. This situation occurs because excessively small inter-event times do not allow the simulator to process all the messages involved with the event e_k before the arrival of the next event, e_{k+1}. When this situation arises, queued unprocessed messages are accumulated, and therefore the simulation presents an evident degradation of performance. The degradation of performance can be noticed by observing the *worst-case response* time for a given simulation. Here, a simulation with an *inter-event period* of *20 milliseconds* results in a *worst-case response time* of *15260 milliseconds*. In contrast, when the *inter-event period* is *180 milliseconds*, the *worst-case response time* is reduced to *20 milliseconds*.

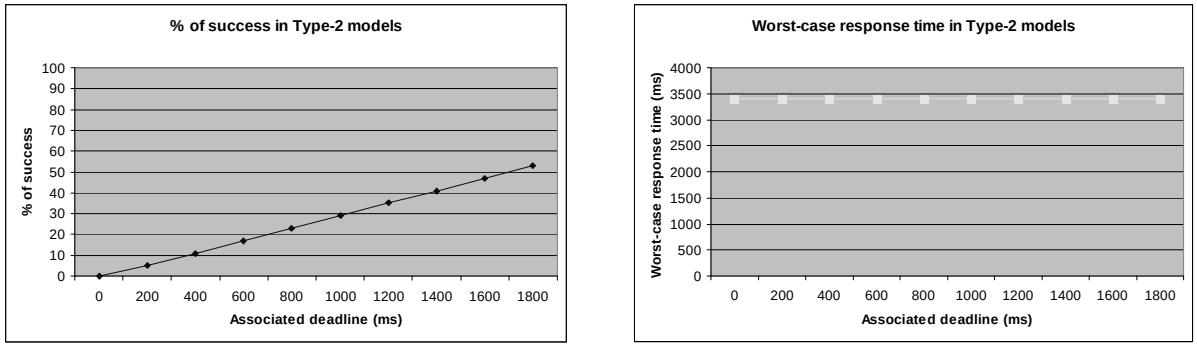
4.4.4.2 Varying the associated deadlines for models without workload

In the following simulations, Type-2 models are employed and inter-event periods remain stable. Here, the associated deadlines vary from 0 to 1800 milliseconds. These cases show how the strictness of deadlines affects the *percentage of success*.

Simulation parameter	Associated value
Number of components per level	7 components
Number of levels in the hierarchy	10 levels
Model type	Type-2
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	50 ms
Associated deadlines	0 - 1800 ms
Number of atomic components in the obtained models	55
Number of coupled components in the obtained models	9

Table 13: Simulation parameters – Varying associated deadlines, Type-2

The following figure shows the obtained results for these models.



(a) *(b)*

Figure 42: Real time execution of Type-3 models with varying associated deadlines (a) Percentage of success, (b) Worst-case response time

Figure 42 (a) shows how the strictness of the deadlines can impact on the *percentage of success* for a given simulation. Extremely tight timing constraints do not allow the simulator to meet those deadlines on time. As deadlines become more relaxed, the *percentage of success* is correspondingly increased because constraints are more likely to be met.

In contrast, *Figure 42 (b)* shows that *worst-case response times* remain constant, regardless of variations on the associated deadline time. Actually, these experiments show that the response time for an event is unrelated to the timing constraint it might have.

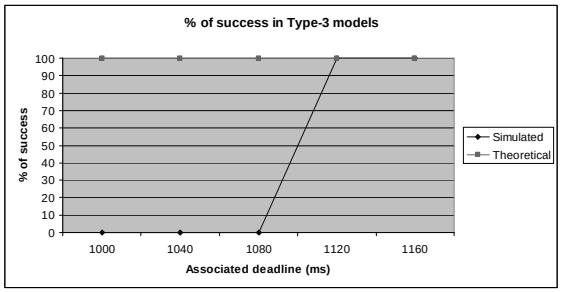
4.4.4.3 Varying the associated deadlines for models with workload

The previous test cases, where the associated deadlines were variable, are repeated here for models with workload in their transition functions. Type-3 models are employed and, again, inter-event periods remain stable. The associated deadlines vary from 1000 to 1160 milliseconds.

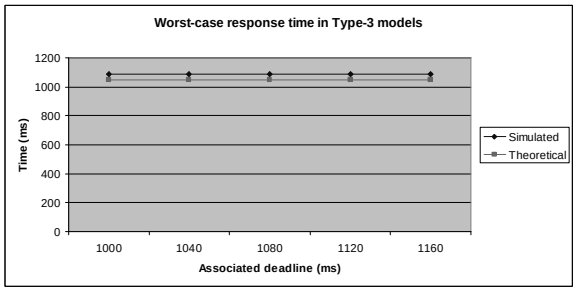
Simulation parameter	Associated value
Number of components per level	6 components
Number of levels in the hierarchy	5 levels
Model type	Type-3
Workload in internal transition function	50 ms
Workload in external transition function	50 ms
Number of external events	100 events
Inter-event period	50 ms
Associated deadlines	1000 – 1160 ms
Number of atomic components in the obtained models	21
Number of coupled components in the obtained models	4

Table 14: Simulation parameters – Varying associated deadlines with workload, Type-3

The following figure shows the obtained results for these models that include workload in their atomic transition functions.



(a)



(b)

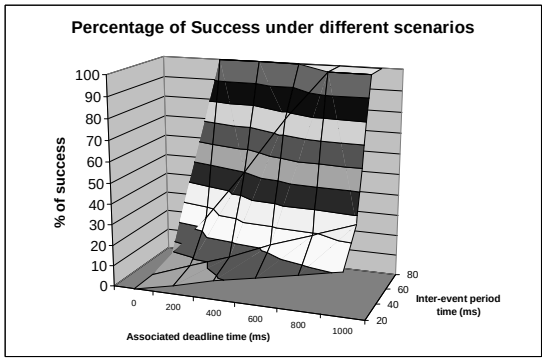
Figure 43: Real time execution of Type-2 models with varying associated deadlines and workload (a) Percentage of success, (b) Worst-case response time

Figure 43 (a) shows that when the associated deadline is approximately greater than 1100 milliseconds we can achieve 100% of success in our simulation, meeting all the associated deadlines.

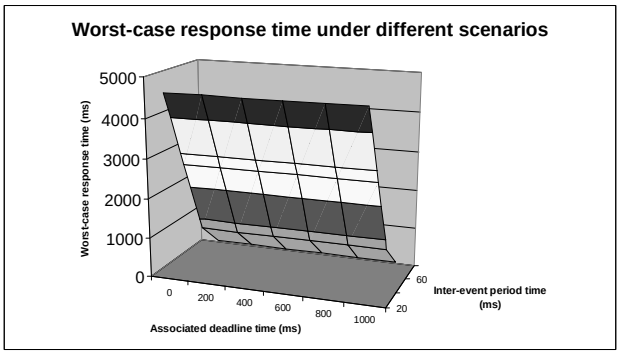
As we have stated earlier in this section, *Figure 43 (b)* illustrates that the *worst-case response time* remains stable regardless of the associated deadline. In addition, this chart shows that when we execute models with time-consuming code in their transition functions, the incurred overhead is relatively small and it does not affect seriously the *worst-case response times*.

4.4.4.4 Combination of inter-event period times and associated deadlines

The previous subsections have studied how the frequency of event arrival and the strictness of the constraints can affect the simulation of a given model separately.
The following charts combine the previous experiments and provide further information about the simulation under different scenarios.



(a)



(b)

Figure 44: Real time execution of Type-3 models under different scenarios (a) Percentage of success, (b) Worst-case response time

These charts show the obtained results after the execution of several simulations performed under a combination of different frequencies of event arrival and different strictness on the imposed deadlines.

Figure 44 illustrates not only the favorable impact of greater inter-event period times (*i.e.* lower frequencies of events), but also greater associated deadlines (*i.e.* less strict constraints).

Even with deadlines that are not very strict (1000 milliseconds) if the inter-event period is 20 milliseconds, only 20% of the deadlines are met. As deadlines become more relaxed, the percentage of success is correspondingly increased because constraints are more likely to be met, regardless of the frequency of events.

Furthermore, we can point out that with fair constraints and frequencies, simulation results are correct. For example, when a simulation is executed using 600 milliseconds in deadlines and 60 milliseconds between event arrivals, a success of 77% is achieved. If the same frequency of events is received by a simulation whose deadlines are 800 milliseconds, then the success is 100%.

4.4.5 Varying workload in transition functions

We have shown how the variation of several parameters may affect the results of real time model execution. This subsection describes the effect of executing different workload on the transition functions, while the other parameters remain unchanged.

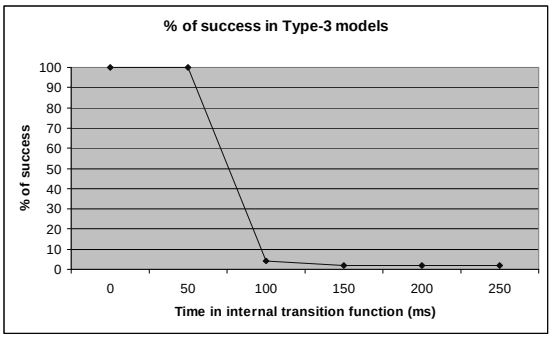
Atomic models can execute time-consuming code in both their internal and external transition functions. Recall from Chapter 2 that our synthetic model generator produces Dhrystone code [Wei84] to resemble real workload that would be executed by the atomic components.

Firstly, we executed simulations using different workload in the internal transition functions. The following table summarizes the parameters used to run simulations with 0 to 250 milliseconds in the internal transition functions.

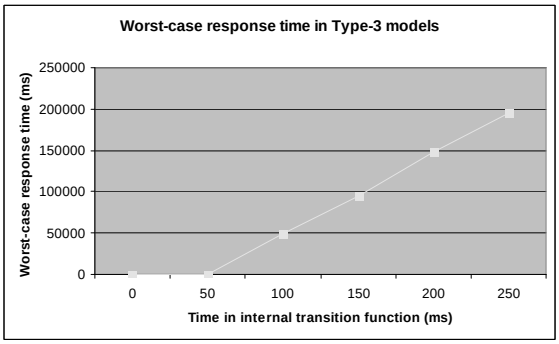
Simulation parameter	Associated value
Number of components per level	4 components
Number of levels in the hierarchy	4 levels
Model type	Type-3
Workload in internal transition function	0-250 ms
Workload in external transition function	0 ms
Number of external events	50 events
Inter-event period	5000 ms
Associated deadlines	5000 ms
Number of atomic components in the obtained models	10
Number of coupled components in the obtained models	3

Table 15: Simulation parameters – Varying time spent in internal transition functions

The results are shown below:



(a)



(b)

Figure 45: Real time execution of Type-3 models with varying time in internal transition functions (a) Percentage of success, (b) Worst-case response time

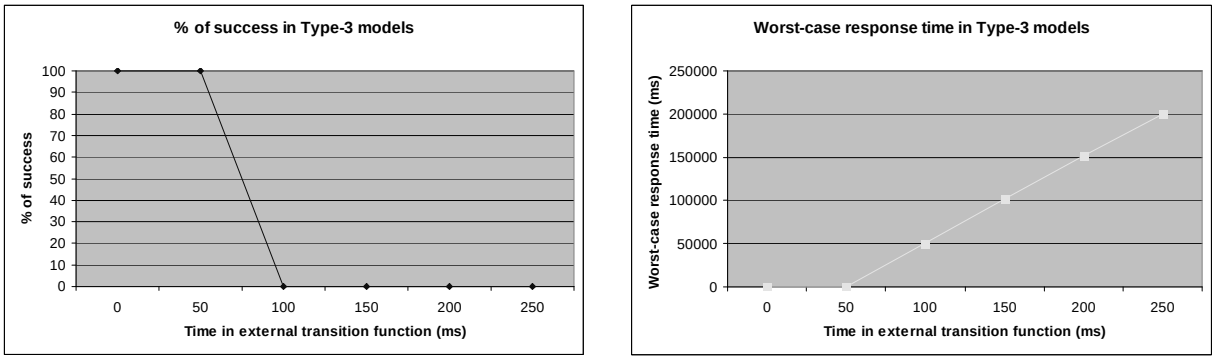
As the workload in the internal transition functions grows, the *percentage of success* is reduced, especially when the time-consuming code is 100 milliseconds or more. Additionally, the *worst-case response times* are evidently increased, because of the time that has to be spent on executing the atomic transition functions.

Secondly, we executed simulations using different workload in the external transition functions.

Simulation parameter	Associated value
Number of components per level	4 components
Number of levels in the hierarchy	4 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0-250 ms
Number of external events	100 events
Inter-event period	10000 ms
Associated deadlines	1000 ms
Number of atomic components in the obtained models	10
Number of coupled components in the obtained models	3

Table 16: Simulation parameters – Varying time spent in external transition functions

The results are shown in the next figure:



(a) *(b)*

Figure 46: Real time execution of Type-3 models with varying time in external transition functions (a) Percentage of success, (b) Worst-case response time

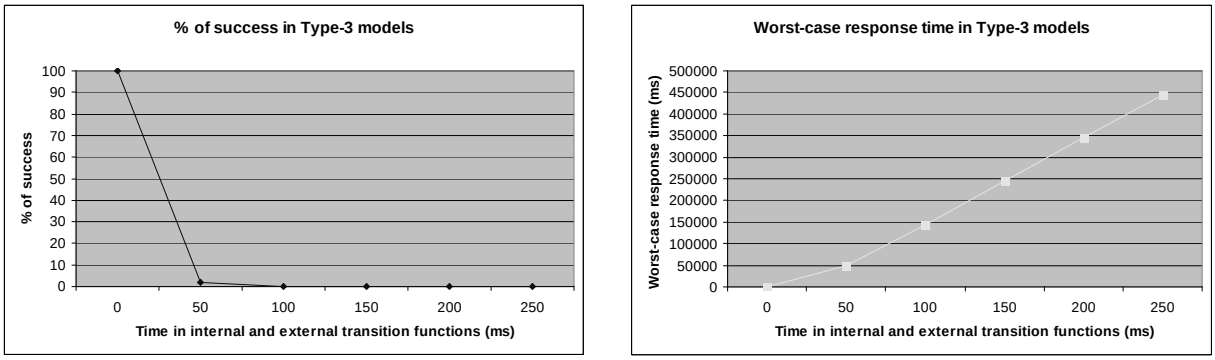
Figure 46 shows analogous results to the ones obtained using varying time in internal transition functions.

Lastly, we executed simulations using different workload in both transition functions. The first model does not execute workload in the internal transition function, neither in the external transition function. The second model executes 50 milliseconds in the internal transition function and 50 milliseconds in the external transition function. The third case executes 100 milliseconds in each function, and so on. Finally, the sixth model executes 250 milliseconds in each transition function.

Simulation parameter	Associated value
Number of components per level	4 components
Number of levels in the hierarchy	4 levels
Model type	Type-3
Workload in internal transition function	0-250 ms
Workload in external transition function	0-250 ms
Number of external events	100 events
Inter-event period	10000 ms
Associated deadlines	1000 ms
Number of atomic components in the obtained models	10
Number of coupled components in the obtained models	3

Table 17: Simulation parameters – Varying time spent in internal and external transition functions

The next figure shows the results for these experiments:



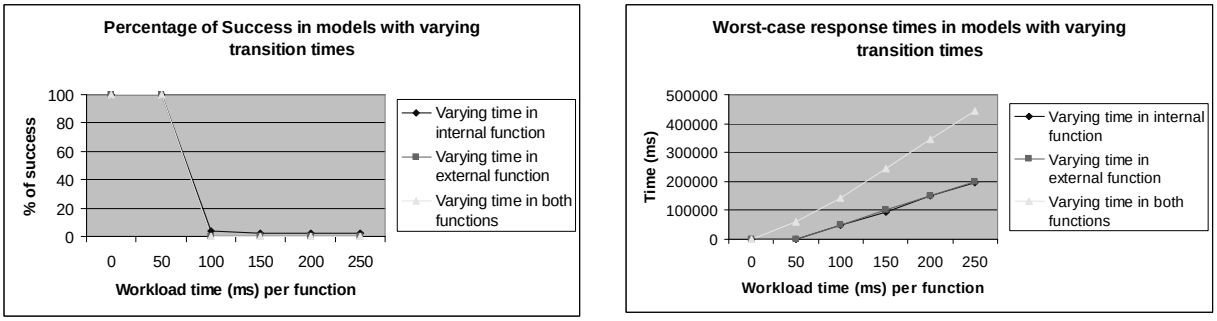
(a) *(b)*

Figure 47: Real time execution of Type-3 models with varying time in internal and external transition functions (a) Percentage of success, (b) Worst-case response time

Again, the charts are quite similar to those illustrated above. Here, the *worst-case response times* are greatly increased because of the large amount of code that has to be executed in both transition functions.

4.4.5.1 Combination of results for variable time in transition functions

The following chart shows the previous results combined in two charts in order to provide a comparison. It is possible to observe the *percentage of success* and *worst-case response time* for executions with time in the internal transition function, in the external transition function, and in both functions conjointly.



(a) *(b)*

Figure 48: Real time execution of Type-3 models with varying time in their transition functions (a) Percentage of success, (b) Worst-case response time

The figures show proper *percentages of success* when the workload time per function is 0 or 50 milliseconds, in spite of the place where the time-consuming code is being executed. In contrast, a noticeable reduction in the *percentage of success* is observed in all cases when the time in the transition functions is increased to 100 milliseconds.

In general, as the workload in the transition functions grows, the *percentage of success* is reduced in all the experiments.

Additionally, *Figure 48 (b)* shows that the *worst-case response times* are evidently increased, because of the time that has to be spent on executing the atomic transition functions. When the workload is executed in both transition functions, the *worst-case response time* is doubled.

4.4.6 Execution of large-scale models

Sometimes, models may have much larger sizes than those employed in the previous subsections. Our intention is to provide a testing analysis that includes such types of models. The following cases show some samples of very large models.

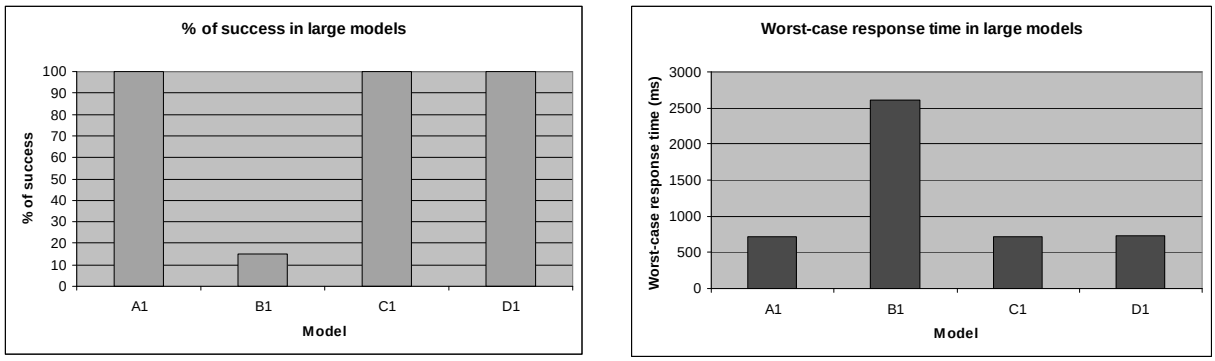
4.4.6.1 Type-1 large models

The first experiments employed Type-1 simple models. The parameters are as follows,

	Simulation			
Simulation parameter	**A1**	**B1**	**C1**	**D1**
Number of components per level	100 components	150 components	200 components	400 components
Number of levels in the hierarchy	100 levels	75 levels	50 levels	25 levels
Model type	Type-1	Type-1	Type-1	Type-1
Workload in internal transition function	0 ms	0 ms	0 ms	0 ms
Workload in external transition function	0 ms	0 ms	0 ms	0 ms
Number of external events	20 events	20 events	20 events	20 events
Inter-event period	1000 ms	1000 ms	1000 ms	1000 ms
Associated deadlines	1000 ms	1000 ms	1000 ms	1000 ms
Number of atomic components in the obtained models	9802	11027	9752	9577
Number of coupled components in the obtained models	99	74	49	24

Table 18: Simulation parameters – Large models (Type-1)

It is important to point out that the obtained Type-1 models have approximately ten thousand components in their structures. Therefore, the overhead needed to carry out the simulation is greatly increased. Because of this characteristic, not only the frequencies of events are lower but also the associated deadlines are less strict.



(a) *(b)*

Figure 49: Real time execution of large-scale Type-1 models: (a) Percentage of success, (b) Worst-case response time

Figure 49 shows the results for large Type-1 models. *Worst-case response times* are relatively small if we consider the model size. However, Type-1 models are very simple, and models that are more complex must be simulated to provide further results about execution performance.

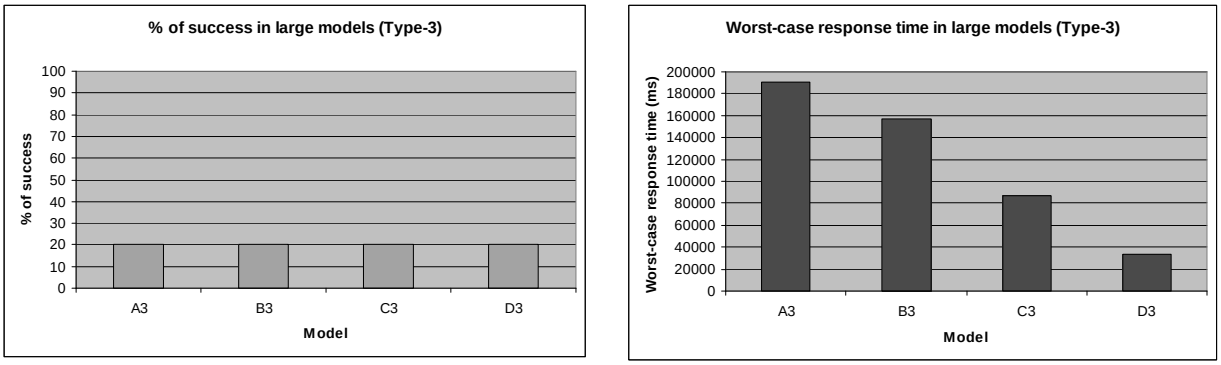
4.4.6.2 Type-3 large models

The previous testing has been repeated with much more complex models, which belong to the Type-3 group. Due to their increased number of interconnections, these models execute many more internal and external transition functions in response to an incoming event (see Chapter 2 for further details). Consequently, the intervals between events are increased and the associated deadlines are noticeable less strict.

	Simulation			
Simulation parameter	**A3**	**B3**	**C3**	**D3**
Number of components per level	100 components	150 components	200 components	400 components
Number of levels in the hierarchy	100 levels	75 levels	50 levels	25 levels
Model type	Type-3	Type-3	Type-3	Type-3
Workload in internal transition function	0 ms	0 ms	0 ms	0 ms
Workload in external transition function	0 ms	0 ms	0 ms	0 ms
Number of external events	5 events	5 events	5 events	5 events
Inter-event period	5000 ms	5000 ms	5000 ms	5000 ms
Associated deadlines	5000 ms	5000 ms	5000 ms	5000 ms
Number of atomic components in the obtained models	9802	11027	9752	9577
Number of coupled components in the obtained models	99	74	49	24

Table 19: Simulation parameters – Large models (Type-3)

The obtained results are shown in the next figure.



(a) *(b)*

Figure 50: Real time execution of large-scale Type-3 models: (a) Percentage of success, (b) Worst-case response time

Figure 50 illustrates the *percentages of success* and the *worst-case response times* that result from the execution of large-scale Type-3 models. Lower frequencies and less restrictive deadlines allowed the models to be executed, even though percentages of success and response times are far from being optimal. Results show that the overhead incurred by simulating these complex large models becomes much more noticeable, especially when the models are very deep (*i.e.* the number of levels in the hierarchy is high). Degradation of performance is an undesirable consequence of the execution of these extremely large and complex models.

The worst-case response times are larger than expected, essentially because of queued messages that are still pending from previous external events. When a new external event is received in this situation, then the response is greatly delayed due to the increased number of messages that still have to be processed by the simulator. If the simulation receives events with lower frequency (*i.e.* the inter-event period is larger), then better response times can be achieved.

This phenomenon of accumulating unprocessed messages can occur on any kind of model. However, large complex models are more prone to experiencing these problems.

4.5 Conclusions about performance analysis using the real time simulator

The real time extension of the toolkit was tested using a wide variety of models. We executed small, medium and large models to show the behavior and limitations of CD++ under several circumstances. Different model complexities have been used.

Moreover, different timing constraints and environments have been studied. The impacts of both the frequency of event arrival and the strictness of the associated deadlines have been analyzed.

The analysis shows adequate performance on most cases, with response times that are quite reasonable for the executed models. Nevertheless, missed deadlines and poor response times may occur if the tool experiences an important degradation of performance.

We have shown that performance degradation is a consequence of extremely large (or complex) model structures, excessive high frequency on event arrivals or immoderate strictness on the imposed deadlines. Particularly, the accumulation of unprocessed messages is an essential factor that affects performance when the frequency of event arrival is high.

Performance degradation that results from the execution of large and complex models is inherent to the simulation methodology. As we have explained before, the simulation technique is based on the exchange of messages between simulators and coordinators. The message-passing process may impact on the execution

performance, mainly if the model structure is too large or complex. The next chapter describes the implementation of a flattened simulation mechanism to reduce the performance degradation in such cases.

5. FLATTENED SIMULATION TECHNIQUE

Earlier, in Chapter 2, we provided an in-depth analysis of the simulator performance using the *virtual time* approach. Additionally, the previous section described a thorough testing of the *real time* extension. Even though results are appropriate in most cases, it is desirable to provide a more efficient simulator.

We explained that a *real time* simulation usually interacts with its surrounding environment, and the system must deliver an answer before a certain deadline. When the execution performance becomes poor, responses cannot be produced on time and therefore deadlines are missed. A more efficient real time simulator would allow achieving better results on more complex scenarios.

Not only the *real time* approach, but also the *virtual time* approach can take advantages of a more efficient simulator. Recall that when the *virtual time* approach is employed, inactivity periods are skipped. Therefore, *virtual time* simulations can evolve faster, and even faster outcomes might be obtained if a more efficient simulator is provided. In conclusion, a reduction of the execution simulation time can also benefit the user of the *virtual time* technique.

First, we explain the problems that may arise when the hierarchical simulation approach is employed. Later, the design and implementation of a new flattened simulator is presented.

5.1 Problems of the hierarchical simulation approach

The previous sections showed appropriate execution results on most cases. Nevertheless, as the size and complexity of models grows, a reduction of performance becomes more noticeable. The main reason for this loss of performance is the overhead incurred by the exchange of messages between simulators and coordinators, which serves as a basis for carrying out the simulation.

Recall from the first chapter the concordance between models and DEVS processors that is shown in the next figure:

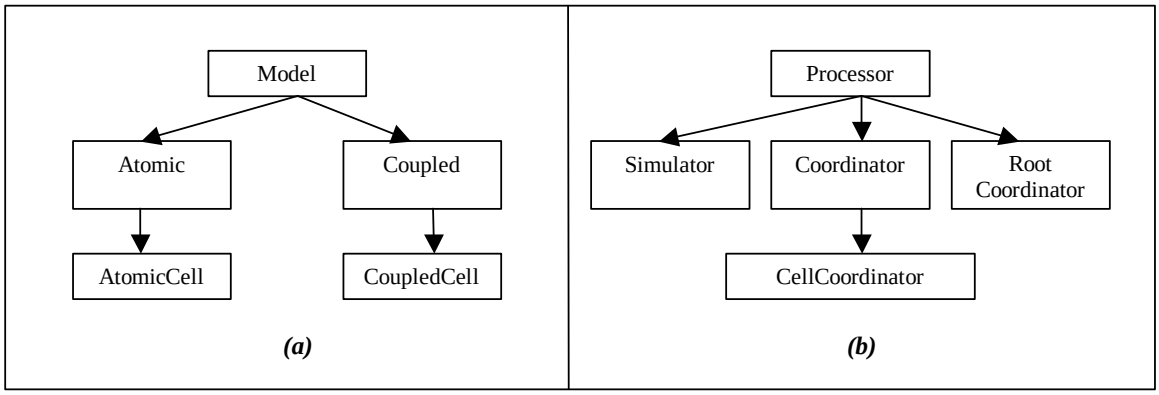


Figure 51: CD++ (a) Model hierarchy, (b) Processor hierarchy

Figure 51 (a) shows the classes that are involved with the atomic and coupled components in DEVS and Cell-DEVS models. *Figure 51 (b)* shows the classes that are involved with the simulation technique.

When a DEVS model is executed, one *simulator* object is created for each *atomic* component. On the other hand, one *coordinator* object is created for each *coupled* component in the hierarchy. The same idea holds when a Cell-DEVS model is executed. In this case, a *simulator* object is created for each existing cell, whereas a *CellCoordinator* is created for each Cell-DEVS model.

The function of a *simulator* is to manage its associated *atomic* component. It executes the δ_{int}, δ_{ext} and λ(s) functions. In contrast, a *coordinator* object manages an associated *coupled* component and the port mapping of its inner components.

In addition to the components described above, one *root coordinator* is created to manage global aspects of the simulation. It is directly involved with the topmost-coupled component, which has the highest level in the model hierarchy. The *root coordinator* maintains the global time, and it starts and stops the simulation process. Lastly, it receives the output results that must be sent to the environment.

The following figure shows a sample model with a few components:

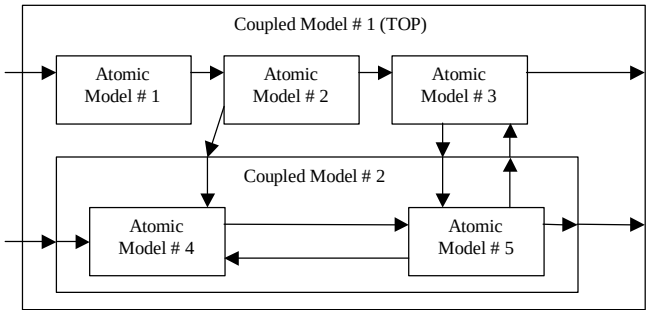


Figure 52: Sample model structure

The figure shows a sample model whose topmost component has three atomic submodels (*Atomic Models #1*, *#2* and *#3*) and one coupled model (*Coupled Model #2*). The inner-coupled component is formed by two atomic components (*Atomic Models #4* and *#5*).

The following figure shows the model hierarchy and the corresponding processor hierarchy obtained in CD++ when the hierarchical simulation is used.

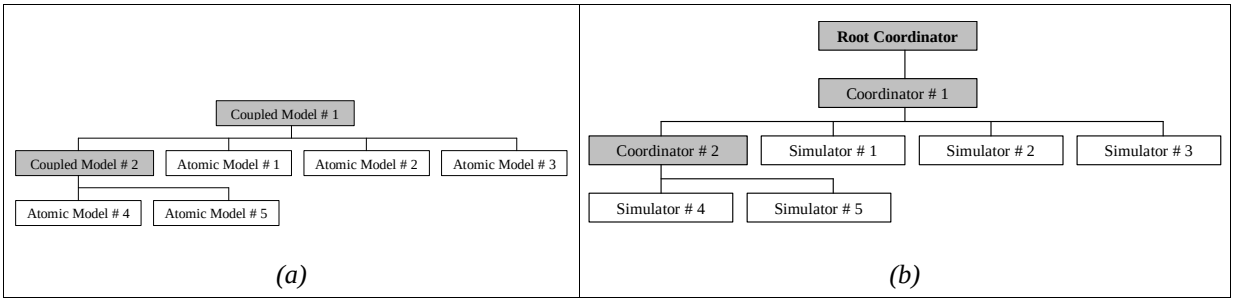


Figure 53: Hierarchical simulation approach: (a) Example of a model hierarchy, (b) Associated processor hierarchy obtained in the example

The figure shows that the model hierarchy is rather replicated in the processor hierarchy, using *coordinators* instead of *coupled* components, and *simulators* instead of *atomic* components. It also shows the *root coordinator* added on top of the hierarchy.

Each time the *root coordinator* has to schedule an event to lowermost simulators (*Simulators #4* and *#5*), the overhead incurred by message passing can be considerable. Firstly, the *root coordinator* has to send a message to the *Coordinator #1*. Secondly, the *Coordinator #2* forwards this message to the *Simulators #4* and *#5*. Only then, the simulators are able to execute the transition function of their associated atomic models. A similar phenomenon is produced if the *Simulator #5* sends an output through a port connected to *Simulator #3*. The number of intermediate coordinators can be arbitrarily high depending on the studied model.

As we have pointed out before, the simulation process is message driven; it is based on the message exchange among *simulators*, *coordinators* and the *root coordinator*. Messages contain information to identify the *sender* and the *receiver*. A *time-stamp* for the message and an associated *value* are also included in the packet.

The following table illustrates the number of simulators and coordinators created for some sample models, and the number of messages involved with the processing of a single external event, along with other parameters in hierarchical simulations.

	Simulation			
Simulation parameter	**A1**	**A3**	**B1**	**B3**
Number of components per level	100 components	100 components	150 components	150 components
Number of levels in the hierarchy	100 levels	100 levels	75 levels	75 levels
Model type	Type-1	Type-3	Type-1	Type-3
Number of atomic components	9802	9802	11027	11027
Number of simulators	9802	9802	11027	11027
Number of coupled components	99	99	74	74
Number of coordinators	99	99	74	74
Number of root-coordinators	1	1	1	1
Number of messages exchanged to process a single external event	79220	3484718	89416	2958468

Table 20: Examples employing a hierarchical simulation approach

Models **A1** (Type-1) and **A3** (Type-3) have the same number of atomic and coupled components; therefore they have an equal number of *simulators* and *coordinators*. They only differ in the number of interconnections within their inner components, which is given by the *Model type*. This difference results in the remarkably increased quantity of messages needed to process a single event in **A3**.

If models **B1** (Type-1) and **B3** (Type-3) are compared, the same differences can be observed.

When the simulated models are larger, the number of *atomic* and *coupled* components is increased. Then, as we have shown, the number of *simulators* and *coordinators* grows accordingly. Due to the message-passing technique among processors, the incurred overhead grows with the number of existing simulators and coordinators, and the degradation of performance becomes noticeable.

Table 20 shows that almost 90000 messages have to be exchanged in order to entirely process a single event in these Type-1 models, whereas up to 3500000 are exchanged in similar Type-3 models.

The new simulation technique addresses this problem by reducing the number of messages exchanged in the simulation process.

5.2 Implementation of the flattened simulation technique

The main problem to be resolved is the overhead incurred by message passing among processors. To overcome this issue, a new flattened simulation technique was implemented in the CD++ toolkit. The approach is called

flattened in contrast to the *hierarchical* one previously explained. A similar development for other DEVS simulator can be found in [Kim00].

The DEVS formalism separates the model from the actual abstract simulator. The new flattened simulator keeps this important property, so only the simulation mechanism is revised. Then, the model class hierarchy is unchanged. On the other hand, the DEVS processor hierarchy is extended. A new *flattened coordinator* that inherits from the *processor* class is created. The complete class hierarchy is shown in the next figure.

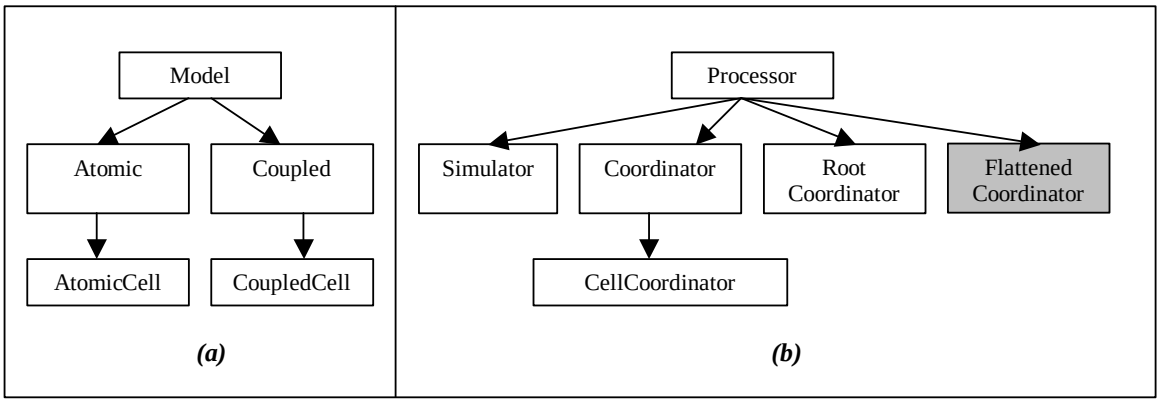


Figure 54: CD++ (a) Model hierarchy (unchanged)
(b) Extended processor hierarchy (includes the new Flattened Coordinator)

Figure 54 shows the extended processor hierarchy with the new *Flattened Coordinator* (shaded box) that is derived from *Processor* class.

The new technique creates only two processors to execute a simulation. The first processor is the usual *root coordinator* that still manages global aspects of the simulation. The second processor that is created is the new *flattened coordinator*, which was designed to perform the tasks of *simulators* and *coordinators*. No other processor is created in order to carry out the simulation.

The following table shows the processors that are involved in a non-hierarchical simulation using the same sample models than before.

	Simulation			
Simulation parameter	**A1**	**A3**	**B1**	**B3**
Number of components per level	100 components	100 components	150 components	150 components
Number of levels in the hierarchy	100 levels	100 levels	75 levels	75 levels
Model type	Type-1	Type-1	Type-3	Type-3
Number of atomic components	9802	9802	11027	11027
Number of simulators	**0**	**0**	**0**	**0**
Number of coupled components	99	99	74	74
Number of coordinators	**0**	**0**	**0**	**0**
Number of root-coordinators	1	1	1	1
Number of flattened coordinators	**1**	**1**	**1**	**1**

Table 21: Examples employing a flattened simulation approach

Table 21 shows that when the flattened simulation approach is used, the number of *simulators* and *coordinators* is zero regardless of the number of *atomic* and *coupled* components in the models. However, one *flattened coordinator* is created to provide the complete functionality of all *simulators* and *coordinators*.

To execute a flattened simulation using neither coordinators nor simulators, the following items have been resolved:

- Due to the absence of a *simulator* linked to each *atomic* model, now the *flattened coordinator* executes the δ_{int}, δ_{ext} and λ(s) functions for each *atomic* component. It also stores the information about the *time of next transition* (t_N), *time of last transition* (t_L) and the *external events* that are queued for each *atomic* component.

- The *flattened coordinator* must transform the hierarchical structure of the model to a flattened structure in order to reduce the overhead incurred by message passing. The resulting non-hierarchical structure is used by the flattened coordinator.

- Due to the absence of the usual coordinators, now the flattened coordinator maps the ports for all atomic and coupled components in the model hierarchy. Then, the component links are handled by the *flattened coordinator*, which forwards the events as needed.

- The *flattened coordinator* must receive and send messages directly with the root coordinator in order to carry out the simulation process.

If we simulate the model described in the previous subsection (see *Figure 52*) using the flattened approach, the resulting hierarchy is remarkably simplified and the overhead incurred by message passing is significantly reduced. Both the model and the processor hierarchies are shown in the next figure.

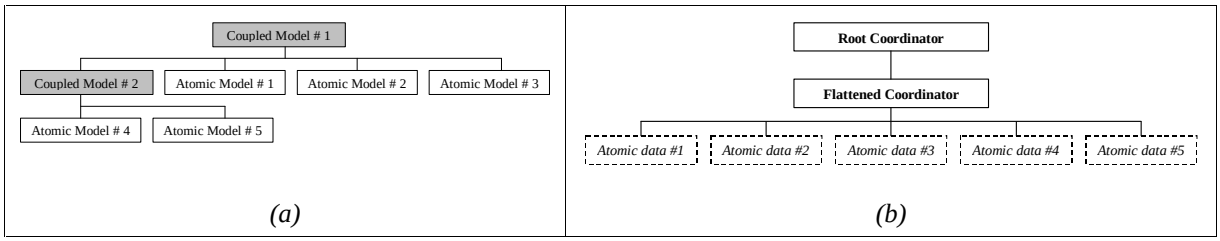


Figure 55: Flattened simulation approach.
(a) Example of a model hierarchy, (b) Associated processor hierarchy

When the flattened simulation technique is used, the associated processor hierarchy is greatly simplified. Messages are exchanged only between the *root coordinator* and the *flattened coordinator*, the only two processors that are created in the new hierarchy. The dotted boxes represent the *atomic data*, which are objects created to store the information about each atomic model. The *flattened coordinator* updates the *atomic data* and handles the execution of the *transition* and *output functions* for each atomic component without any intermediate processor.

5.3 Conclusions about the flattened simulation technique

The existing techniques in CD++ use a hierarchical simulation approach. The design and implementation of the new *flattened coordinator* presented in this chapter allows the execution of simulations using a non-hierarchical approach.

The reduction of messages when the flattened approach is used can boost performance results, especially when the incurred overhead is related to the model size and complexity. The new flattened technique can be used not only for *virtual time* but also for *real time* simulation on DEVS and Cell-DEVS models.

The next chapter presents a comparison between the flattened and hierarchical approaches using both *virtual time* and *real time*.

6. PERFORMANCE ANALYSIS OF THE FLATTENED SIMULATOR

In order to assess the efficiency of the new flattened approach, we present an analysis of the developed simulator. Not only DEVS but also Cell-DEVS models are executed.

The results obtained by means of the flattened simulator are compared with those obtained with the hierarchical approach.

Analysis of both *real time* and *virtual time* simulations are provided, using not only synthetically generated models but also existing ones from the CD++ library.

6.1 Test notes

The testing described in this chapter was performed in the **ParDEVS Laboratory**, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. All simulations were run on a *Compaq ProLiant 1600* node, consisting of a Pentium II 450MHz processor with 512 MB of RAM, 512-KB second-level ECC cache and 100-MHz GTL Bus.

The installed operating system was *Caldera OpenLinux*.

6.2 Virtual time execution analysis

In this subsection, executions of both flattened and hierarchical techniques are compared using the virtual time simulation. The main goal of the new flattened simulator in virtual time simulations is to reduce the execution time, providing the results faster than the usual approach.

6.2.1 Synthetically generated DEVS models

This subsection provides a comparison of the flattened and hierarchical techniques using the synthetic model generator.

6.2.1.1 Varying number of levels in the hierarchy

The first series of models have a fixed number of components per level, and a variable number of levels in the hierarchy (*depth*).

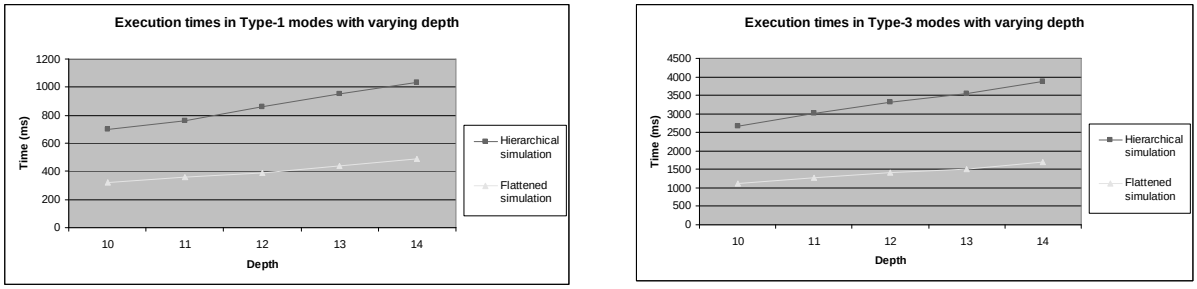
6.2.1.1.1 Models without workload in the transition functions

Firstly, models will not execute workload in their transition functions. Therefore, overhead is compared more easily. The following table shows the parameters corresponding to this test.

Simulation parameter	Associated value
Number of components per level	8 components
Number of levels in the hierarchy (Depth)	10 to 14 levels
Model type	Type-1 and Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Number of atomic components in the obtained models	73 to 92
Number of coupled components in the obtained models	9 to 13

Table 22: Simulation parameters – Varying depth,Type-1 and Type-3 models without workload

The following figure shows the obtained execution times for each simulation technique in Type-1 and Type-3 models.



(a) *(b)*

Figure 56: Execution time for hierarchical and flattened simulations with varying depth, without workload, (a) Type-1 models, (b) Type-3 models

Figure 56 shows the execution time for Type-1 and Type-3 models, using both the hierarchical and flattened simulators. In these experiments, the width is fixed, the depth is variable and there is no workload in the atomic transition functions.

For instance, a Type-1 model with *10 levels of depth* is executed in 700 milliseconds using the hierarchical approach, whereas only 320 milliseconds are needed to execute the same model using the flattened approach. Analogous results can be observed in Type-3 models.

When the depth is increased, the difference between the hierarchical and flattened execution time becomes more noticeable. Clearly, the flattened simulator outperforms the hierarchical simulator in all the performed experiments.

6.2.1.1.2 Models with workload in the transition functions

The previous examples provided a comparison between hierarchical and flattened simulation of models without overhead. Here, the models have workload in their transition functions. Consequently, we can measure the overhead in cases that execute code in their atomic components.

The simulation parameters are as follows.

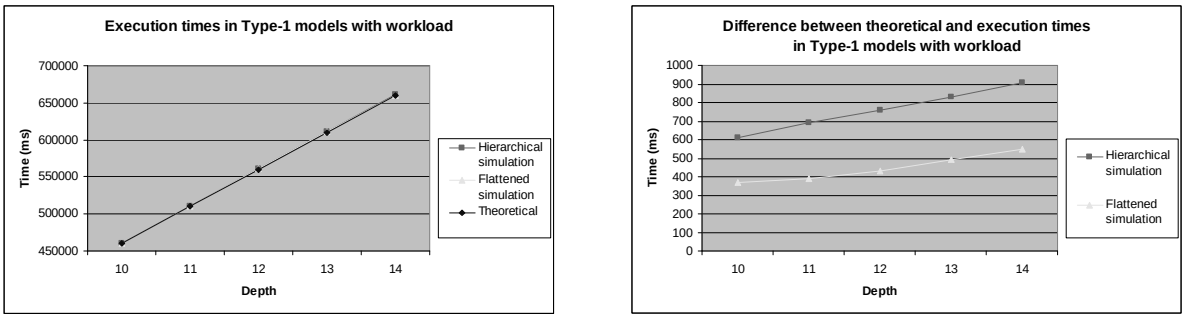
Simulation parameter	Associated value
Number of components per level	6 components
Number of levels in the hierarchy (Depth)	10 to 14 levels
Model type	Type-1
Workload in internal transition function	50 ms
Workload in external transition function	50 ms
Number of external events	100 events
Number of atomic components in the obtained models	46 to 66
Number of coupled components in the obtained models	9 to 13

Table 23: Simulation parameters – Varying depth, Type-1 models with workload

The following chart shows the execution time for both simulation techniques. In addition, the theoretical *execution time* is included in the chart. As we have explained earlier, the *theoretical execution time* for a given simulation does not include any overhead at all. It is the sum of all time spent in executing internal and external transition functions all along this simulation. It can be measured as follows,

Total theoretical time = [(# External Transitions * TimeInExternalTransition) + (# Internal Transitions * TimeInInternalTransition)] * NumberOfEvents

The *theoretical time* can be compared with the obtained execution times for both the hierarchical and flattened simulation techniques. Moreover, the differences between the theoretical and the execution time for each technique are described in an additional chart.



(a) *(b)*

Figure 57: Type-1 models with varying depth and workload
(a) Execution times, (b) Difference between experiments and theoretical execution times

Figure 57 (a) shows that the execution times employing both the hierarchical and flattened approaches are very similar to the theoretical execution time. The workload executed in the transition functions remarkably increases the total execution time, which reduces the impact of the overhead incurred by both simulators.

However, *Figure 57 (b)* shows the difference between the theoretical execution time and each simulation technique. The execution times for the flattened simulations are lower than the execution times for the hierarchical simulations.

Furthermore, it is possible to measure the *percentage of overhead* incurred by each simulation technique. It is computed by subtracting the theoretical time from the execution time and dividing that by the execution time itself, that is:

$$Overhead\ (\%) = \frac{(executionTime - theoreticalTime)}{executionTime}$$

The following figure presents the overhead incurred by each simulation technique.

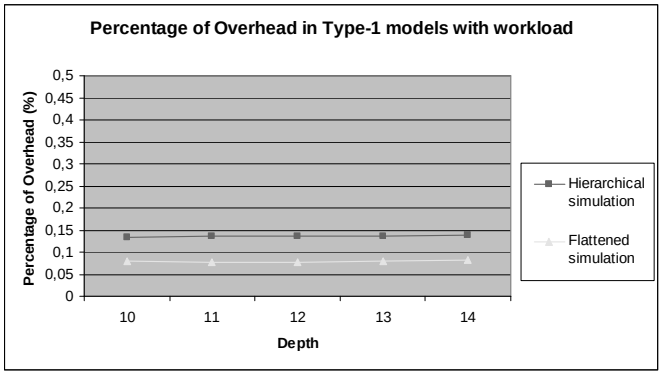


Figure 58: Percentage of overhead incurred by hierarchical and flattened simulators in Type-1 models with varying depth and workload

In general, these executions show a relatively small overhead. Particularly, the figure illustrates that the overhead incurred by the flattened simulator is lower than the overhead incurred by the hierarchical one.

In models with workload like those executed in this subsection, the proposed flattened technique provides better performance results and outperforms the existing hierarchical technique.

6.2.1.2 Varying number of components per level

These models have a fixed number of components in the hierarchy, but a variable number of components per level (*width*). The execution of models with and without workload is analyzed.

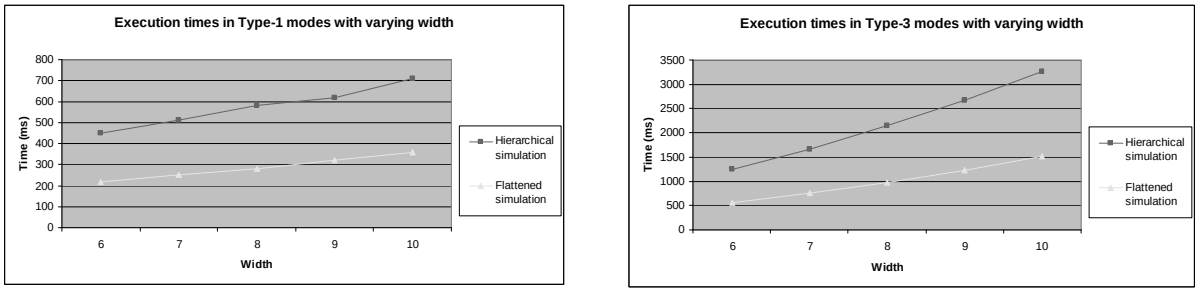
6.2.1.2.1 Models without workload in the transition functions

We start running models that do not execute workload in their transition functions. Therefore, only overhead is executed in these cases. The following table shows the parameters that have been employed.

Simulation parameter	Associated value
Number of components per level	6 to 10 components
Number of levels in the hierarchy (Depth)	8 levels
Model type	Type-1 and Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Number of atomic components in the obtained models	36 to 64
Number of coupled components in the obtained models	7

Table 24: Simulation parameters – Varying width, Type-1 and Type-3 models without workload

The following figure shows the obtained execution times for each simulation technique.



(a) *(b)*

Figure 59: Execution time for hierarchical and flattened simulations with varying width, without workload, (a) Type-1 models, (b) Type-3 models

Figure 59 (a) and *(b)* show that the flattened simulator outperforms the hierarchical simulator in all the executed Type-1 and Type-3 cases in which the width was variable. When the models become larger, the difference between the hierarchical and the flattened approach is more evident.

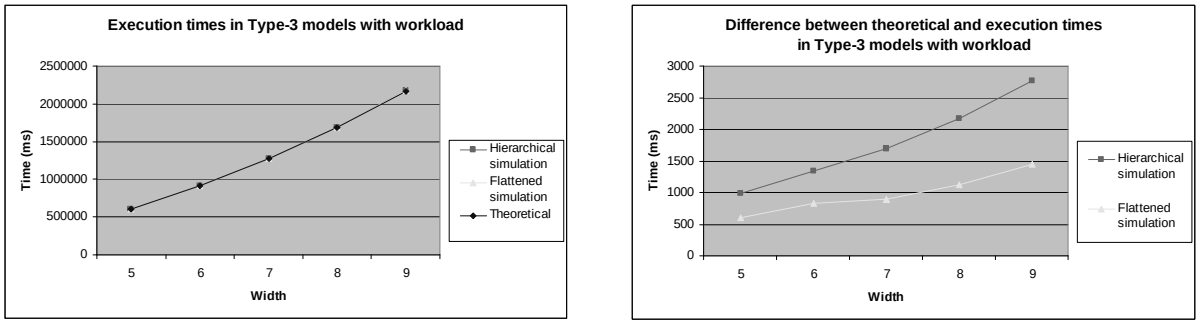
6.2.1.2.2 Models with workload in the transition functions

In addition, experiments have been performed using Type-3 models with workload in their atomic transition functions. The table shows all the parameters.

Simulation parameter	Associated value
Number of components per level	5 to 9 components
Number of levels in the hierarchy (Depth)	7 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Number of atomic components in the obtained models	25 to 49
Number of coupled components in the obtained models	6

Table 25: Simulation parameters – Varying width, Type-3 models with workload

The following charts show the execution times for both techniques, and the difference between the experiments and the theoretical results.



(a) *(b)*

Figure 60: Type-3 models with varying width and workload
(a) Execution times, (b) Difference between experiments and theoretical execution times

The following figure shows the percentages of overhead incurred by each simulation technique.

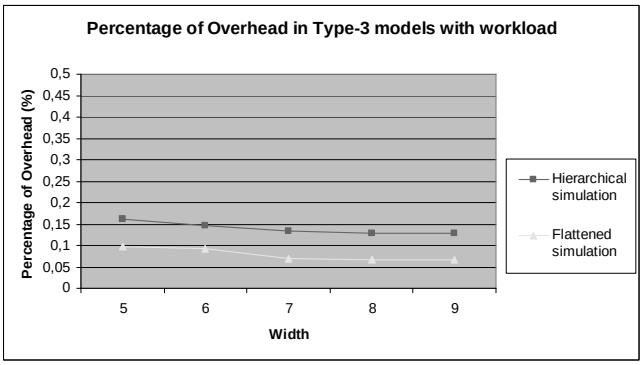


Figure 61: Percentage of overhead incurred by hierarchical and flattened simulators in Type-3 models with varying width and workload

The figure shows that the overhead remains stable for both simulation techniques. Actually, the overhead might be reduced because of the increased amount of workload executed in the transition functions of larger models.

The flattened approach outperforms the hierarchical approach in all these experiments. The use of the non-hierarchical approach can reduce the overhead in 50%, therefore providing better execution times.

6.2.1.3 Large Type-1 models

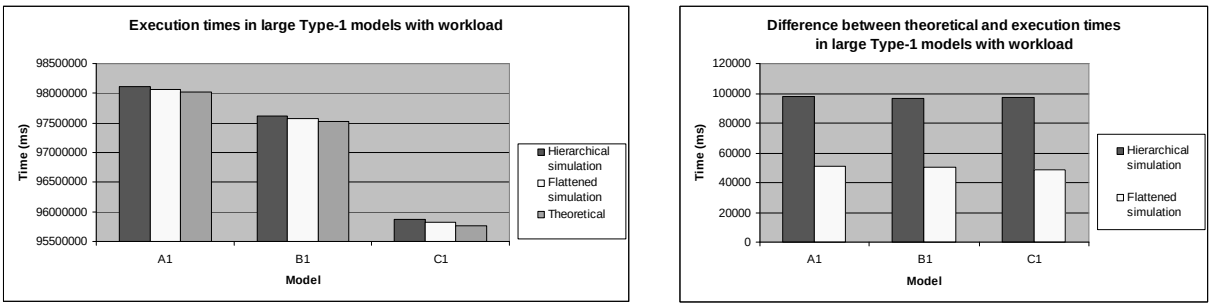
As we have explained earlier, models may have much larger sizes than those employed in the previous subsections. The following cases show some samples of very large models.

	Simulation		
Simulation parameter	**A1**	**B1**	**C1**
Number of components per level	100 components	200 components	400 components
Number of levels in the hierarchy	100 levels	50 levels	25 levels
Model type	Type-1	Type-1	Type-1
Workload in internal transition function	50 ms	50 ms	50 ms
Workload in external transition function	50 ms	50 ms	50 ms
Number of external events	100 events	100 events	100 events
Number of atomic components in the obtained models	9802	9752	9577
Number of coupled components in the obtained models	99	49	24

Table 26: Simulation parameters – Large models (Type-1)

The obtained Type-1 models have almost ten thousand atomic components in their structures. Consequently, the overhead needed to carry out the simulation is considerable.

The following charts show the execution times for both hierarchical and flattened simulators, and the difference between the experiments and the theoretical results.



(a) *(b)*

Figure 62: Large Type-1 models with workload
(a) Execution times, (b) Difference between experiments and theoretical execution times

These simulations take a long time to be executed. The model A1, for instance, took more than 90000000 milliseconds (approximately 27 hours) to be entirely executed. However, the differences between the theoretical execution time and the experiments are quite small; less than 97790 milliseconds (approximately one minute and

37 seconds) if the hierarchical simulator is used and less than 50920 milliseconds (approximately 51 seconds) if the flattened simulator is used.

The following figure shows the percentages of overhead incurred by each simulation technique in these large models.

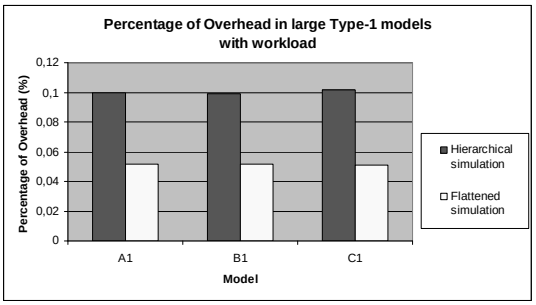


Figure 63: Percentage of overhead incurred by hierarchical and flattened simulators in Large Type-1 models with workload

Even though the execution times are considerable, the obtained overheads are quite small. In all cases, the flattened simulator is more efficient than the hierarchical simulator. *Figure 63* shows that the overheads are reduced in almost 50% in these large Type-1 models when the flattened technique is employed.

6.2.1.4 Large Type-3 models

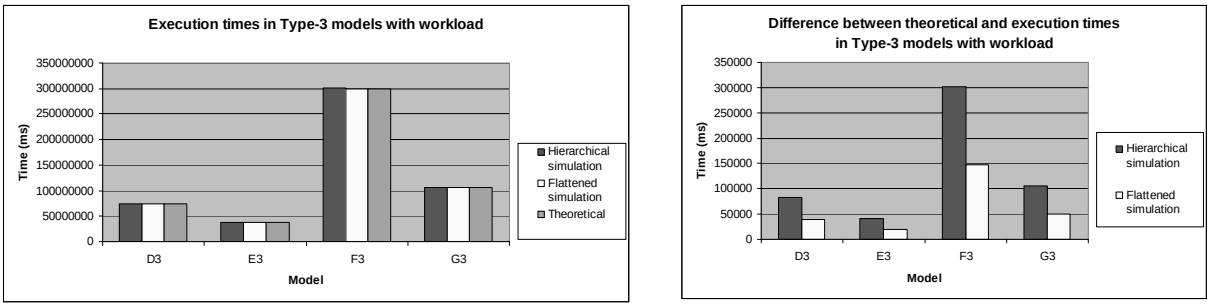
The previous section studied the execution of large Type-1 models. As we have explained earlier in Chapter 2, Type-1 models are simple and have a small number of interconnections between components. Type-3 models, which have a larger number of interconnections between components, are more complex and are analyzed in this subsection.

Simulation parameter	D3	E3	F3	G3
Number of components per level	20 components	40 components	50 components	30 components
Number of levels in the hierarchy	40 levels	20 levels	50 levels	50 levels
Model type	Type-3	Type-3	Type-3	Type-3
Workload in internal transition function	50 ms	50 ms	50 ms	50 ms
Workload in external transition function	50 ms	50 ms	50 ms	50 ms
Number of external events	50 events	50 events	50 events	50 events
Number of atomic components in the obtained models	742	742	2402	1422
Number of coupled components in the obtained models	39	19	49	49

Table 27: Simulation parameters – Large models (Type-3)

Notice that Type-3 models are more complex than the equivalent Type-1 models. Consequently, the message passing needed to carry out these simulations is considerable and even greater than in Type-1 experiments.

The next figures show the execution times for both simulators, and the difference between the experiments and the theoretical results.



(a) *(b)*

Figure 64: Large Type-3 models with workload
(a) Execution times, (b) Difference between experiments and theoretical execution times

The difference between the theoretical and the execution time is smaller when the flattened simulator is employed. In all cases, the flattened technique is more efficient and provides better execution times.

The following figure shows the percentages of overhead incurred by each simulation technique in these large Type-3 models.

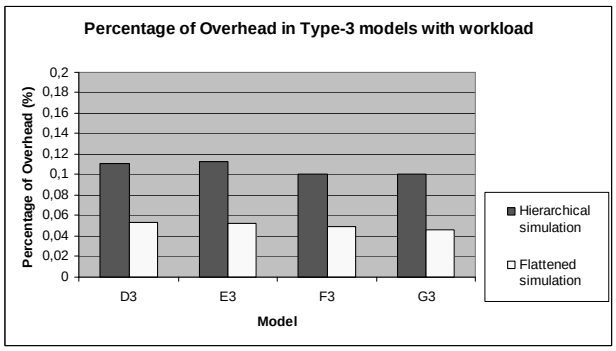


Figure 65: Percentage of overhead incurred by hierarchical and flattened simulators in Large Type-3 models with workload

Again, the overhead is smaller for the flattened simulator in all the experiments. Approximately a 50% of the overhead can be reduced if the non-hierarchical technique is employed to simulate these synthetically generated models.

6.2.2 Existing DEVS models

The previous subsection showed the execution of synthetically generated models. In addition, it is interesting to show the performance of the different simulators executing existing DEVS models.

Previous works have formed an extensive library of DEVS models to be executed in CD++ [Ame00a, Ame00b, Rod99]. Not only simple atomic models, but also very complex coupled models can be found in the library [Wai02].

This section analyzes the execution of such models in CD++ using both the hierarchical and flattened simulators in virtual time.

The following table describes briefly some of the executed models.

	Simulation			
MODEL	**Alarm Clock**	**Elevator**	**GPT**	**FSM**
Brief description	A digital alarm clock with display, alarm, buzzer and snooze.	An elevator with floor buttons to push. It has a door, a control unit and an engine.	The typical Generator-Processor-Transducer model with a queue to buffer processes.	A Moore finite state machine constructed with a base library available in CD++.
Number of coupled components (approx.)	3	3	1	1
Number of atomic components (approx.)	8	4	4	4
Interconnection complexity	Medium	Light-medium	Light	Light

Table 28: Existing models executed in virtual time existing in the CD++ library [Wai02]

The models have a different amount of workload to be executed in the transition functions of their atomic components. In addition, a different interconnection complexity is found in each case. It can be regarded as the quantity and the type of interconnections among the inner components of the model. The table provides an approximate measure of this complexity.

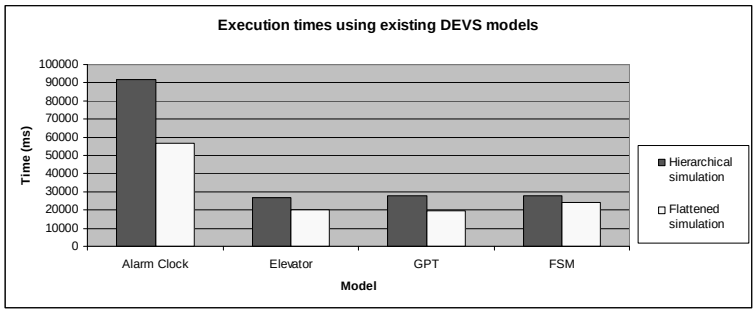


Figure 66: Execution time for existing DEVS models using hierarchical and flattened simulators

In all the cases shown in the figure above, the execution time for DEVS models is reduced when the flattened simulator is used. The next figure shows the percentage of time reduction achieved by the non-hierarchical approach.

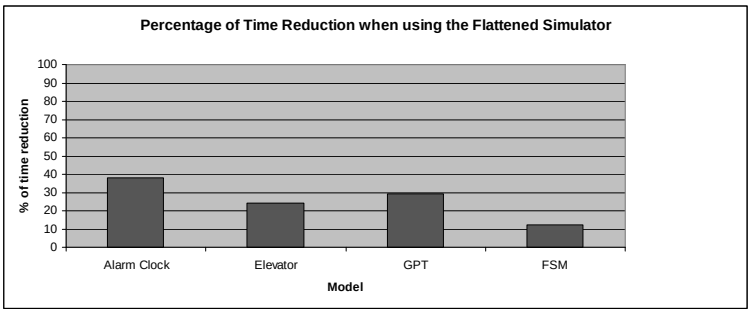


Figure 67: Percentage of time reduction using the flattened approach for existing DEVS models

The percentages of time reduction depend on the DEVS model that is being simulated. In the *Alarm Clock*, we have obtained up to 40% of reductions in the execution time. On the other hand, the *Finite State Machine* has shown time reductions of 10% approximately.

6.2.3 Existing Cell-DEVS models

There are several Cell-DEVS models existing in the CD++ library [Wai02]. The available models include forest fires, life game, heat diffusion, robot movement, colonies of ants and watershed analysis, among others. For more information about Cell-DEVS models, see [Ame00a, Ame00b, Wai02].

It is possible to combine more than one Cell-DEVS component to form a new model. In addition, DEVS models can also be linked to Cell-DEVS models.

The flattened simulator allows the execution of Cell-DEVS models in a non-hierarchical fashion. The execution of non-hierarchical simulations of Cell-DEVS models can be particularly interesting. Usually, because of the large number of messages exchanged between the cell simulators and the coordinator, the amount of time needed to perform cellular model simulations can be extremely long.

An analysis of the execution time of several cellular models is provided in this subsection.

6.2.3.1 Life game

The popular life game can be simulated as a Cell-DEVS model [Ame00a, Ame00b]. In such system, each position represents a cell, which can be either dead or alive. The following figure compares the execution time for both hierarchical and flattened simulations employing different model sizes.

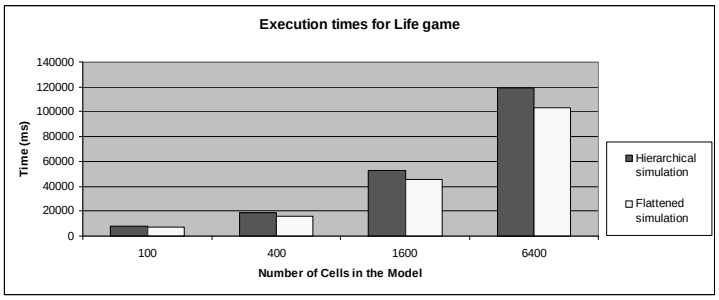


Figure 68: Execution time for Life game in CD++ [Ame00a] using hierarchical and flattened techniques

In every case, the flattened approach outperforms the hierarchical one. The following figure illustrates the percentage of time reduction obtained when the flattened simulator is used.

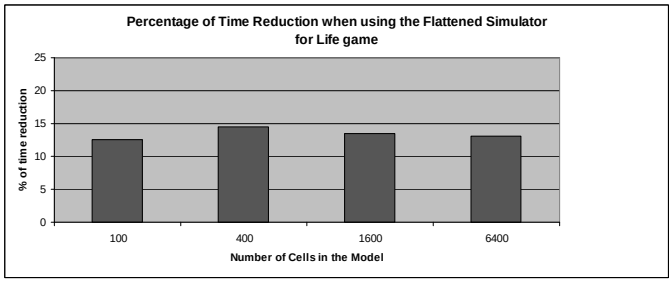


Figure 69: Percentage of time reduction using the flattened approach for Life game in CD++

The previous figure shows that employing the non-hierarchical simulator can reduce the execution time of the Life model up to 15%.

6.2.3.2 Other Cell-DEVS models

There is a wide set of models existing in the CD++ model library [Wai02]. They can be executed, modified or even combined to form new DEVS models. Some of these previously developed models are used here to compare the performance of the different simulation techniques. The following is a brief description of such models:

- The **watershed model** represents a hydrology system built as a cell space [Ame00a]. It is represented as small cells organized in several layers (air, surface water, soil, ground water, and bedrock). The rainfall input is partially retained by vegetation, and the rest infiltrates gradually in the layers.

- A model of **particles of gas** has been simulated using Cell-DEVS. The model simulates fluids of gas moving in different directions. The collision of particles is also defined.

- A **heat diffusion** model studies the spread of heat in a surface [Ame00b]. A Cell-DEVS model represents the surface itself. It is also composed of a heat generator and a cold generator, both of them specified as DEVS models.

- A **colony of ants** has been defined as a Cell-DEVS model. Different ants exist in the same space, trying to find food with random patterns of movement.

- A **substance classifier** is formed by one DEVS and two Cell-DEVS models. A *substance generator* places a given amount of substance into a *queue* component. A *classifier* takes the substance and measures its purity. Then, the product is classified as *first-class* or *second-class*.

- The **movement of sharks** in the sea can be simulated as a Cell-DEVS model. A shark moves following a certain pattern, and may also have contact with smaller fishes to eat in the area.

- The classical **bubble sort** algorithm can be analyzed using a Cell-DEVS model. In this model, each cell represents a numeric value in an array of fixed size. The entire space can be ordered comparing each cell with its neighbors.

- Binary **linear automata** have been defined as Cell-DEVS models. They consist of a series of very simple rules over their neighbors.

These models have been executed using both the hierarchical and flattened simulators. The following figure shows the execution times for each simulation technique using the described models.

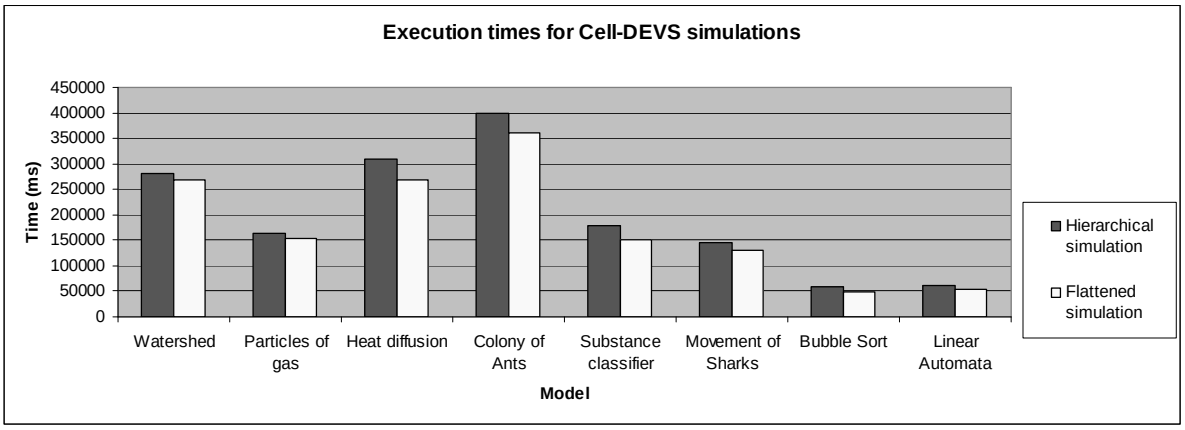


Figure 70: Execution time for Cell-DEVS models using hierarchical and flattened techniques

The previous figure illustrates that the execution time is reduced when the flattened simulator is employed. The following figure shows the percentage of time reduction when the new approach is used.

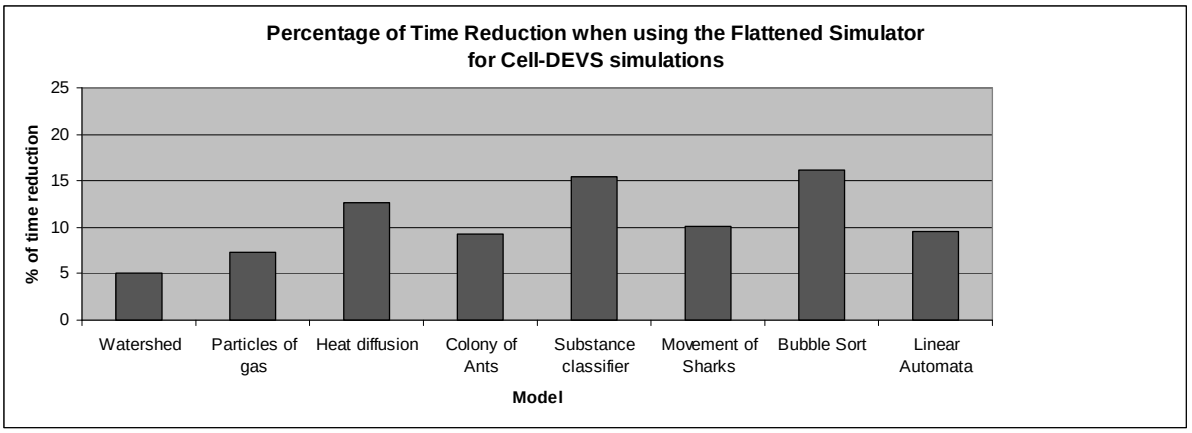


Figure 71: Percentage of time reduction using the flattened approach for Cell-DEVS models

These samples provide meaningful results. The models have several complexities in their structures, and models with different workload have been executed. The reductions in execution time range from 5% to 15% approximately.

6.3 Real time execution analysis

In this subsection, real-time executions of both flattened and hierarchical simulators are compared. Some models that have been tested in Chapter 4 are executed again using the non-hierarchical approach in order to analyze its efficiency.

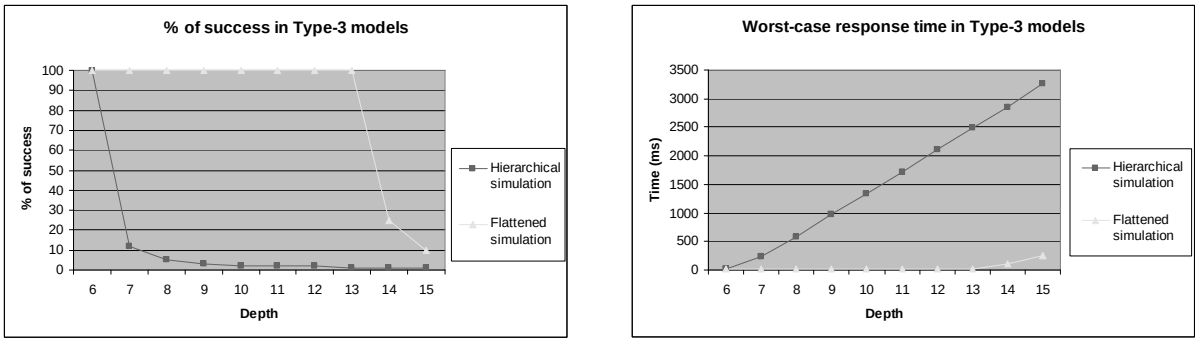
6.3.1 Varying number of levels in the hierarchy without workload

The first experiments show the results that are obtained with varying depth in Type-3 models without workload. The following table summarizes the information corresponding to this test.

Simulation parameter	Associated value
Number of components per level	9 components
Number of levels in the hierarchy (Depth)	6 to 15 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	20 ms
Associated deadlines	20 ms
Number of atomic components in the obtained models	41 to 113
Number of coupled components in the obtained models	5 to 14

Table 29: Parameters for comparison between hierarchical and flattened approaches Varying depth, Type-3 models without workload

The **percentages of success** and the **worst-case response times** are compared for both techniques: *flattened* and *hierarchical*.



(a) *(b)*

Figure 72: Comparison of real-time executions using hierarchical and flattened techniques with varying depth. (a) Percentage of success, (b) Worst-case response time

The figure shows clearly that the *flattened simulation* outperforms the hierarchical simulation in these experiments. Particularly, Type-3 models with several depths (*7* to *13*) are executed with *100%* of success using the *flattened simulation*, while the use of the *hierarchical approach* achieves less than *15%* of success under the same conditions. The *worst-case response times* are greatly reduced when using the flattened approach.

6.3.2 Varying number of levels in the hierarchy with workload

The previous models did not execute workload in their transition functions. The following experiment studies models with time-consuming code in the transition functions.

In addition, the frequency of events depends on the structure of each model. The time between events in each case considers the theoretical time that is needed to process a single event. For example, if the theoretical time needed to entirely process a single event is 1000 milliseconds, then the inter-event period is 1000 milliseconds. In contrast, if the time needed to entirely process an event is 1600 milliseconds, then the period between events is 1600 milliseconds. Thus, the environment for each model depends on the model itself.

Simulation parameter	Associated value
Number of components per level	4 components
Number of levels in the hierarchy (Depth)	4 to 10 levels
Model type	Type-1
Workload in internal transition function	50 ms
Workload in external transition function	50 ms
Number of external events	10 events
Inter-event period	Theoretical execution time for a single event (500 to 2800 milliseconds)
Number of atomic components in the obtained models	10 to 28
Number of coupled components in the obtained models	3 to 9

Table 30: Parameters for comparison between hierarchical and flattened approaches Varying depth, Type-1 models with workload

The following figure shows the theoretical *worst-case response time*. As we have explained earlier in this work, the theoretical results are simply the sum of all the time spent in executing the workload that is found in the internal and external transition functions. Neither the overhead incurred by the simulator nor any other factors that may affect simulation performance are included in the **theoretical** results.

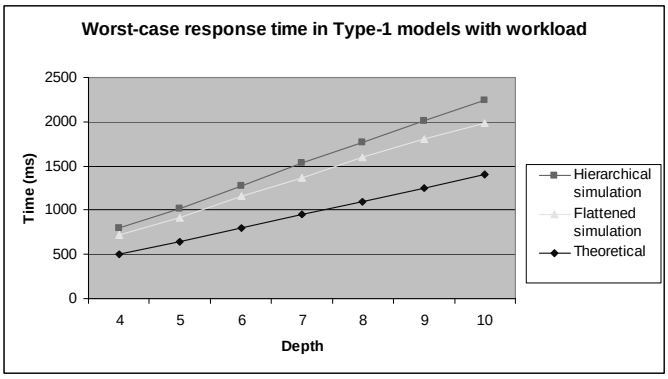


Figure 73: Comparison of worst-case execution time using hierarchical and flattened techniques Type-1 models with variable depth and workload

Figure 73 shows that the use of the flattened simulation technique provides better response times. In deeper models, the difference between the hierarchical and flattened simulators becomes more noticeable.

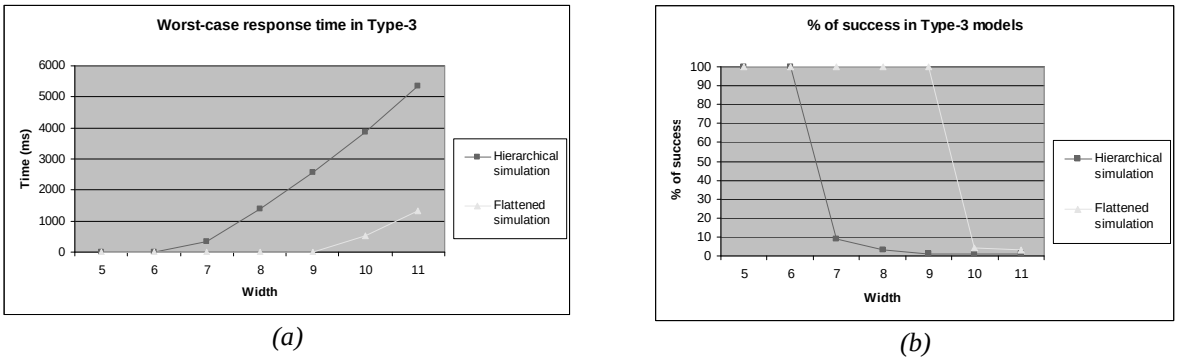
6.3.3 Varying number of components per levels in the hierarchy without workload

The previous experiments have analyzed models whose depth was variable. The following table summarizes the parameters used to test models where the width is variable. There is no workload executed in these experiments.

Simulation parameter	Associated value
Number of components per level	5 to 11 components
Number of levels in the hierarchy (Depth)	6 levels
Model type	Type-3
Workload in internal transition function	0 ms
Workload in external transition function	0 ms
Number of external events	100 events
Inter-event period	30 ms
Associated deadlines	30 ms
Number of atomic components in the obtained models	21 to 51
Number of coupled components in the obtained models	5

Table 31: Parameters for comparison between hierarchical and flattened approaches Varying width (components per level), Type-3 models

The following charts illustrate the obtained results for these Type-3 models without workload.



(a) *(b)*

Figure 74: Comparison of real-time executions using hierarchical and flattened techniques with varying width. (a) Percentage of success, (b) Worst-case response time

Figure 74 shows that the flattened simulator outperforms the hierarchical one, providing better response times and greater percentages of success. The use of the non-hierarchical simulator allows the execution of larger models with better performance results.

6.3.4 Varying number of components per levels in the hierarchy with workload

These experiments have a fixed depth and a varying number of components per level with workload. Again, the frequency of events depends on the structure of each model. Hence, the time between events is equal to the theoretical time that is needed to process a single event.

Simulation parameter	Associated value
Number of components per level	3 to 10 components
Number of levels in the hierarchy (Depth)	7 levels
Model type	Type-1
Workload in internal transition function	50 ms
Workload in external transition function	50 ms
Number of external events	10 events
Inter-event period	Theoretical execution time for a single event (650 to 1850 milliseconds)
Number of atomic components in the obtained models	13 to 37
Number of coupled components in the obtained models	6

Table 32: Parameters for comparison between hierarchical and flattened approaches

The next figure shows the theoretical *worst-case response time* in addition to the hierarchical and flattened results to compare the results.

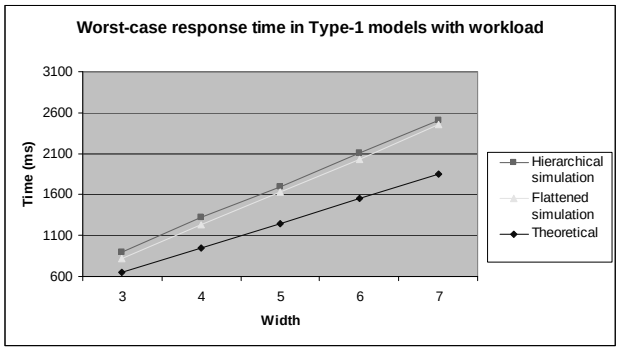


Figure 75: Comparison of worst-case execution time using hierarchical and flattened techniques Type-1 models with variable width and workload

Figure 75 shows a comparison of both simulation approaches. Again, the flattened technique outperforms the hierarchical one, obtaining lower *worst-case response times* in all the experiments.

6.4 Conclusions about the performance of the flattened simulator

We have conducted a thorough testing of the new flattened simulator, comparing the results with those obtained using the hierarchical simulator. Both synthetically generated models and existing models from the CD++ library were executed. The experiments included *virtual time* and *real time* model execution.

When the *virtual time* approach is used, in most cases the flattened simulator is more efficient and reduces the simulation time. On the other hand, when the *real time* approach is used, the flattened simulator provides better response times and greater percentages of success.

Not only DEVS but also Cell-DEVS models have been executed employing the new simulation technique. When the flattened simulator is used, the processor structure is more simple and, usually, more effective.

The use of the non-hierarchical simulator reduces the number of messages exchanged in the simulation process. This reduction of overhead leads to better performance results. In general, we have shown that the new flattened simulator outperforms the hierarchical one.

7. CONCLUSIONS

Testing the performance of a simulator is usually a very complicated task. We have developed a synthetic model generator to facilitate the testing phase. The tool produces DEVS models that are similar to those existing in the real world. Models with different sizes and shapes can be easily generated. To emulate several degrees of complexity in their structures, three different types of models have been defined. In addition, it is possible to determine a given workload to be executed in the atomic transition functions. The workload is Dhrystone code that resembles the tasks to be performed by the atomic components.

A thorough testing has been carried out on different simulation techniques provided in the CD++ toolkit. The performance of each simulator has been characterized. The overhead incurred by the different simulators is bounded and the performance is appropriate in most cases. The obtained results have shown the possibility of developing a real time extension to the toolkit.

The real time extension to the toolkit has been entirely developed. In such extension, events must be handled timely and time constraints can be stated and validated accordingly. The real time simulator ties the advance of the simulation-time to a wall-clock time (*i.e.* physical time). Consequently, these new features would allow interaction between the simulator and the surrounding environment. The new real time simulator has been tested and analyzed.

The benchmark experiments have shown good results on real time executions. We have studied the percentage of success and worst-case response times under different scenarios. Several properties of the model and its environment have been analyzed. Some weaknesses have been pointed out in the analysis of the tool, specifically on the execution of extremely large models. The message-passing process may impact on the execution performance, mainly if the model structure is too large or complex. Even though the performance degradation was small, it was desirable to provide more efficiency not only in real time but also in virtual time simulations. Thus, a new flattened simulator has been presented to overcome the described problems.

The flattened simulator transforms the hierarchical structure of a model to a flattened structure in order to reduce the overhead incurred by the message passing among simulators and coordinators. The resulting non-hierarchical structure is more simple and more effective. The non-hierarchical approach can be applied not only for DEVS but also for Cell-DEVS simulations.

A thorough testing has been performed to the flattened simulator. In most cases, the flattened technique outperforms the hierarchical technique.

8. REFERENCES

[Ame00a] Ameghino, J.; Wainer. G. “Application of the Cell-DEVS paradigm using N-CD++”. In *Proceedings of the SCS Summer Multiconference on Computer Simulation*. Vancouver, Canada. 2000.

[Ame00b] Ameghino, J.; Wainer. G. “Modelling complex cellular models using N-CD++”. *Master’s thesis.* Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 2000. (in Spanish).

[Cho94] Chow, A.; Ziegler, B. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism”. In *Winter Simulation Conference Proceedings*. SCS, Orlando, USA. 1994.

[Cho98] Cho, S. M.; Kim, T. G.; “Real-Time DEVS simulation: concurrent time-selective execution of combined RT-DEVS and interactive environment”. 1998 Summer Computer Simulation Conference, pp. 410-415. Reno, Nevada, USA. 1998.

[Cho00] Cho, S. M.; Kim, T. G.; “Real-Time Simulation Framework Based on RT-DEVS Formalism”. In *Proceedings of the International Conference on Information Systems, Analysis and Synthesis.* Orlando, USA. 2000.

[Jac01] Jacques, C. “Modelling and simulation of an alarm clock in CD++”. *Internal report*. Department of Sciences and Computer Engineering, Carleton University. Ottawa, ON, Canada. 2001.

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

[Li01] Li, L. “Modelling and simulation of an vending machine in CD++”. *Internal report*. Department of Sciences and Computer Engineering, Carleton University. Ottawa, ON, Canada. 2001.

[Mar97] Martin, D.; McBrayer, T.; Radhakrishan, R.; Wilsey, P. “Time Warp Parallel Discrete Event Simulator”. *Technical report*. Computer Architecture Design Laboratory. University of Cincinnati. USA. 1997.

[Rod99] Rodriguez, D.; Wainer, G. “New extensions to the CD++ tool”. In *Proceedings of SCS Summer Multiconference* on Computer Simulation. Chicago, USA. 1999.

[Sta88] Stankovic J.; “Misconceptions about real time computing: A serious problem for next generation systems”. *IEEE Computer*, Vol. 21, No. 10, pp. 10-19, October 1988.

[Sta96] Stankovic J.; “Strategic Directions in Real-Time and Embedded Systems”. *ACM Computing Surveys*, 50th Anniversary Issue, Vol. 28, No. 4, pp. 751-763, December, 1996.

[Tro01a] Troccoli, A.; Wainer, G. “CD++, a tool for simulating Parallel DEVS and Parallel Cell DEVS models”. *Technical report.* Departamento de Computación, Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 2001.

[Tro01b] Troccoli, A.; Wainer, G. "Performance results of parallel Cell-DEVS execution". In *2001 Summer Computer Simulation Conference*. Orlando, USA. 2001.

[Wai98] Wainer, G.; Giambiasi, N. "Specification, modeling and simulation of timed Cell-DEVS spaces". Technical Report n.: 98-007. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 1998.

[Wai00] Wainer, G.; "Improved cellular models with parallel Cell-DEVS". *Transactions of the SCS*. June 2000.

[Wai01] Wainer, G.; Barylko, A.; Beyoglonian, J. "Experiences with DEVS modeling and simulation". In *IASTED Journal on Modeling and Simulation*. March 2001.

[Wai02] Wainer, G. "Cell-based discrete event simulation". Available via: <http://www.sce.carleton.ca/faculty/wainer/wbgraf> [accessed May 20, 2002]

[Wei84] Weicker, R. P. "Dhrystone: A synthetic systems programming benchmark". In *Communications of the ACM*, volume 27, pages 1013-1030, 1984.

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

[Zei00] Zeigler, B.; Kim, T.; Praehofer, H. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press. 2000.

APPENDIX A - WEB GRAFLOG: AN APPLET TO VISUALIZE THE RESULTS OF CELL-DEVS SIMULATIONS

This appendix describes a support tool developed to visualize the result of Cell-DEVS simulations. The applet and the complete user's manual can be found in [Wai02].

A.1 Introduction

CD++ allows the execution of both DEVS and Cell-DEVS models. It is particularly interesting to visualize the results of a Cell-DEVS simulation. Several Cell-DEVS models have been simulated using the toolkit, such as urban traffic, forest fires, colonies of ants, robot movement and watershed simulation [Ame00a].

A.2 Obtaining a log file using CD++

In order to view the results of a Cell-DEVS simulation, first we have to store the results of such execution.

Once a Cell-DEVS simulation has been successfully performed, the results can be obtained in a *log file*. The log file stores all the messages (or a certain type of them) that are exchanged along the simulation process. The following table shows a sample execution of the model *Fire.ma* up to the simulation time *02:00:00:000*. In addition, the file *Fire.log* has been specified to store the resulting messages.

```
/home/user/cd++> ./cd++ -mFire.ma -t02:00:00:000 –l > Fire.log
```

Table 33: Execution of a sample Cell-DEVS simulation

In order to visualize the results of a Cell-DEVS simulation, only *Y-messages* (*i.e.* output messages) are needed. The following table exemplifies the use of the *–L* flag, to obtain only a certain type of messages in a simulation. This alternative can reduce the execution time.

```
/home/user/cd++> ./cd++ -mFire.ma -t02:00:00:000 –LY –l > Fire.log
```

Table 34: Execution of a sample Cell-DEVS simulation obtaining only Y-messages

A sample log file for the *Forest Fire* model [Ame00a] is provided in the following figure, as a result of the execution of the previous command.

```
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,0)(02) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,1)(03) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,2)(04) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,3)(05) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,4)(06) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,5)(07) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,6)(08) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,7)(09) / out /      0.00000 / forestfire(01)
Msg: 0 / L / Y / 00:00:00:000 / forestfire(0,8)(10) / out /      0.00000 / forestfire(01)

…

Msg: 0 / L / Y / 01:58:32:139 / forestfire(28,24)(866)/ out / 119.53579 / forestfire(01)
Msg: 0 / L / Y / 01:58:41:812 / forestfire(13,2)(394) / out / 119.69699 / forestfire(01)
Msg: 0 / L / Y / 01:58:41:812 / forestfire(29,18)(890)/ out / 119.69699 / forestfire(01)
Msg: 0 / L / Y / 01:58:57:569 / forestfire(2,2)(64)   / out / 119.95962 / forestfire(01)
Msg: 0 / L / Y / 01:58:57:569 / forestfire(27,27)(839)/ out / 119.95962 / forestfire(01)
Msg: 0 / L / Y / 01:59:40:578 / forestfire(28,10)(852)/ out / 120.67636 / forestfire(01)
```

Table 35: Sample log file

For further information about the execution of simulations with the toolkit, refer to the *CD++ User's Manual* [Wai02].

A.3 Converting a log file using Drawlog

We have shown how to log the messages in a specified file when a simulation is performed. The **Drawlog** application converts these messages in a succession of matrixes stored in a plain-text file. Drawlog is a part of the CD++ toolkit.

The following table shows how to obtain such plan-text file using Drawlog.

```
/home/user/cd++> ./drawlog -mFire.ma -cForestFire -lFire.log -f1 > output.drw
```

Table 36: Execution of Drawlog using a sample log file

The following table shows an excerpt of the obtained output file, after the execution of the previous command. The file is a succession of plan-text matrixes in which each coordinate represents the value of the cell at a given moment in the simulation.

```
Time: 00:00:00:000
     0.000     0.000     0.000     0.000     0.000     …     0.000
     0.000     0.000     0.000     0.000     0.000     …     0.000
     …         …         …         …         …         …     …
     0.000     0.000     0.000     0.000     0.000     …     0.000

…

Time: 01:59:40:578
     0.000     0.000     0.000   117.399   108.871     …    91.813
     0.000     0.000     0.000   114.415   105.887     …    88.829
     0.000     0.000   119.960   111.431   102.902     …    85.845
     …         …         …         …         …         …     …
     0.000     0.000     0.000     0.000     0.000     …     0.000
```

Table 37: Execution of Drawlog using a sample log file

The previous table shows an excerpt of the file that is produced by Drawlog. It shows only a part of the initial and final configurations of cells in the model.

Further information about the use of Drawlog can be found in the *CD++ User's Manual*.

A.4 Visualizing the results graphically

The **web-based Graflog**, which belongs to the CD++ toolkit, has been developed to visualize the results of Cell-DEVS simulations. It can be run with any Java-enabled web browser, like Netscape or Microsoft Internet Explorer. An alternative command-line Graflog can be run under DOS [Ame00b].

The following is a snapshot of the web-based Graflog application.

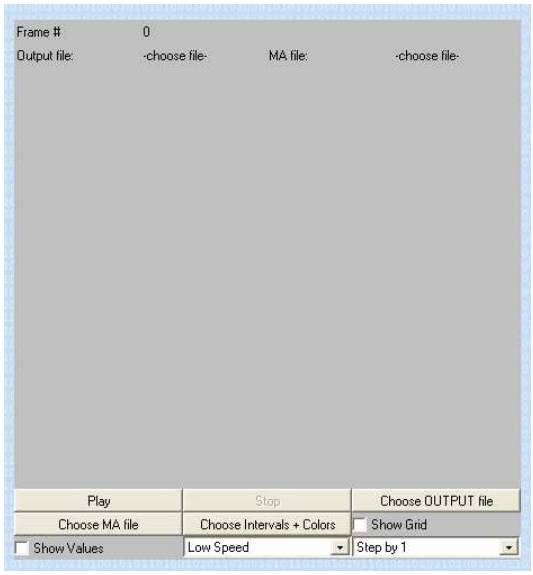


Figure 76: Snapshot of the Web-based Graflog application

The main goal of the Graflog applet is to display graphically the results of a Cell-DEVS simulation. When a Cell-DEVS model is simulated, each cell can take different values along the execution. For instance in a heat diffusion model, each cell represents the temperature of that given coordinate. Alternatively, in a gas diffusion model, each cell can represent the amount of gas in that specific place. The Graflog applet employs the output of the Drawlog application, described earlier in this appendix.

First, we have to specify the color that will represent each interval of values. For example, let us consider a simulation that can take only two possible values; *0* and *1*. Then, it is possible to represent a value of *0* with *black*, and a value of *1* with *white*. However, cells usually take many different values in a simulation. Additionally, the state of a cell can be represented with real numbers. Therefore, several intervals and colors might be chosen by the user in Graflog. Each interval has an associated color that can be chosen from the palette (or entering the RGB composition of the desired color).

The following figure shows the screen where intervals can be defined and colors can be chosen.

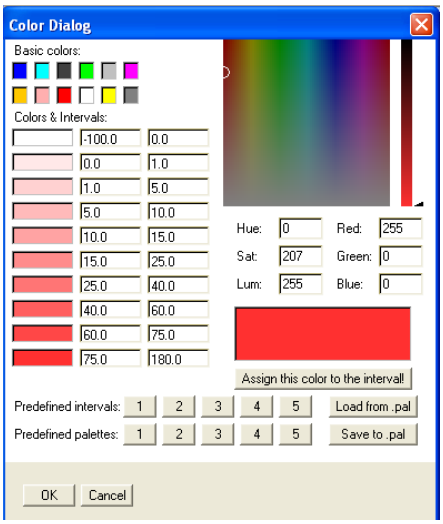


Figure 77: Choosing intervals and associated colors in Web Graflog

The color palette and intervals can be saved to a file. Once each interval has an associated color, we can choose the *output* and *model* files that contain the rest of the information needed to display a Cell-DEVS simulation. Usually, these files have the extensions *.drw* and *.ma* respectively. The *output file* used by Graflog is the result of the execution of the Drawlog application. The *model file* is the information used to execute the simulation with CD++ in the first step.

The following figure shows a sample series of results obtained with the web-based Graflog.

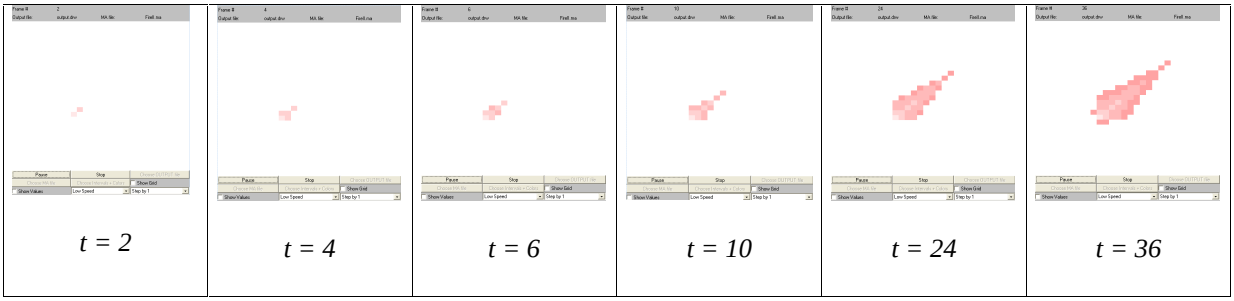


Figure 78: Sample series of results obtained with the toolkit – Forest fire model [Ame00a]

The user can play, pause and repeat the visualization of results. In addition, several aspects of the display can be modified, such as the speed and the step increment.

A.5 Summary of the process

The simulation of Cell-DEVS models can be visualized using the CD++ toolkit. Different applications from the toolkit are used to subsequently format the results. The following figure summarizes the steps needed to visualize such a simulation.

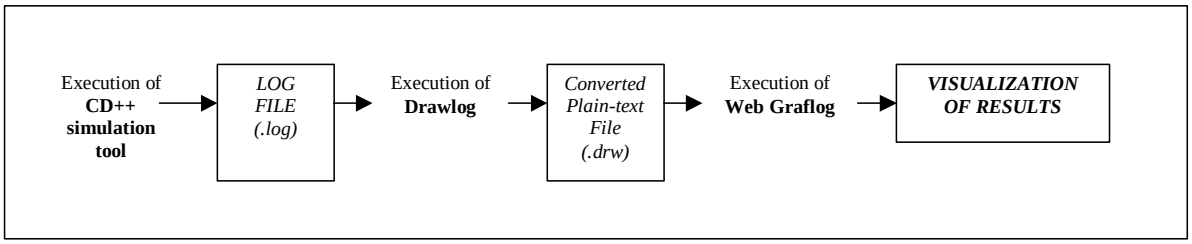


Figure 79: How to visualize the results of a Cell-DEVS simulation using the CD++ toolkit

The complete *Graflog User's Manual* can be found in [Wai02].