Performance modeling of data intensive applications using SystemC

Ganeshan, A.

Award date:
2015

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Performance Modeling of Data Intensive Applications using SystemC

Master Thesis

Ashwath Ganeshan

Supervisor:
dr.L.J.A.M.Lou Somers

océ
A CANON COMPANY

Eindhoven, October 2015

Abstract

This thesis presents the implementation of a Data Intensive Application using SystemC and the implementation was evaluated to find its strength and weakness.

The Data Path of a Printer has the responsibility of transforming an input file (PDF or a post script) into matrices specifying every jet moment whether each nozzle in the print heads of a printer must emit a drop or not. This in turn leads to hundreds of megabits of information, making the data path a suitable candidate as an example system for a data intensive application. The decision of defining and deploying the sub tasks which are responsible for generation and processing of pixels is a difficult task. Therefore, a modelling method based on Y chart methodology was proposed by [5] and a prototype of this method was implemented in Java where the simulation layer was explicitly written.

SystemC is a set of C++ classes and macros which provides a Discrete Event Simulation interface. It was developed to be a design environment for the development of hardware and software. An advantage of SystemC is to model and simulate the systems at a higher abstraction level. SystemC has an inbuilt kernel which takes the responsibility of triggering and executing the processes of the system. In addition to that, SystemC has ports and signals in order to establish communication between different modules in a system. These properties of SystemC were put to use by implementing a data path case of a printer as an example system in SystemC by creating a generic library and measurements were taken in order to benchmark its performance with the same system implemented using C++ and Java.

Though the SystemC simulator performs the entire simulation replacing the simulation layer in C++ and Java, with respect to the simulation speed, SystemC was measured to be nine times slower than Java and almost eighteen times slower than C++. The main reason for the slowdown was the context switching it suffered while running multiple threads parallely. In addition to that, SystemC also suffers a relatively small degradation (up to 10%) in its performance due to the communication overhead caused by ports and signals.

About the Company

océ

A CANON COMPANY

Océ, a Canon Group Company is an international leader in digital document management and printing for professionals. Many Fortune 500 companies and leading commercial printers use Océ solutions for wide format printing, high speed production printing and document related business services. Océ employs 4,000 specialists at innovation and technology centers in Europe, North America and Asia. Through its own Research and Development Océ develops core technologies and majority of its own product concepts. The company was started by 1877 and was involved in coloring dyes and margarines. It was founded by Lodewijk van der Grinten and was later developed by Louis van der Grinten by taking first step of introducing a blueprint for wide format drawings. By 1967, the company entered office printing market and is having its wings over 100 countries today. In 2009, Océ became a Canon company.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Imagining a day without embedded systems in this present generation is impossible. With its presence being felt in several applications such as mobile phones, automobile, energy generation, satellite etc. life can never get more interesting. It is difficult to spot an individual who doesn't use a mobile phone or a residence which doesn't use a washing machine. According to [2], the worldwide market for embedded systems is 160 billion Euros with an annual growth of 9%. With increasing importance, comes the increasing requirements and expectations from the embedded products. Some of the important nonfunctional requirements include cost, speed, power usage, and throughput. In order to meet these requirements, all the embedded devices heavily depend upon the embedded software. The embedded software controls the machines in which it runs on and as like any other software it comes with time and memory constraints. According to a recent study, a Boeing 787 air plane has almost 14 million lines of code; an F-35 fighter jet has almost 24 million lines of code, a car has almost 100 million lines of code giving a brief picture of size and complexity the engineers and designers deal with embedded application development. With performance being an important factor of the application lifetime in the market, *Model Based Performance Analysis* plays an important role in evaluating the performance in the design phase itself to make the product meet its requirements.

1.1 Model Based Performance Analysis

The software life cycle can be briefly divided into three parts. The first phase is where the specifications are written based on the requirements. Depending upon these specifications, the software will be designed and then implemented. After the implementation, the software will be verified for its functionality and also validated to check whether the system meets all the pre-defined requirements. This verification and validation phase plays a crucial role, as the deviation in the behavior from the requirement can be identified and rectified in the early stage itself. The advantage of correcting the behavior in this early stage is to avoid the cost upon the failure of the system when it gets released. The Software architecture in general is a high level description giving the details about the interaction taking place between different subsystems. This description is of two types. One is the static description where the software modules which are necessary for the architecture are identified and the behavior of these modules during run time is given under dynamic description.

This description can be denoted in the Unified Modelling language (UML) , which can be used to visualize a software model in terms of the different subsystems (such as stakeholder, the building blocks, connectors etc.) through structure diagrams and the way in which they communicate using Message Sequence Charts. Since the static design deals only with the subsystems involved, it cannot be used to identify the performance of a system. The performance here refers to metrics such as throughput (which measures the amount of information a system can process), latency (time taken to obtain a response from a system for a given stimulus), energy consumption, power consumption etc. which can be analyzed only through the behavior of the system (dynamic design). After these descriptions, the designers need to come up with an optimal design which in turn must satisfy the specification. But achieving this optimal design is not a easy task. This is because of the fact that systems like automobile, printers; avionics etc comes with a large number of design alternatives. The designers will need a methodology where they can narrow down the number of design steps to be taken. The question that will arise now is what would be those selection criterias that make the design space exploration easier and efficient. In other words, the criteria for the designers to choose a specific architecture over the other must be clear in order to find the optimal design. This is where performance comes into play. The designer can choose an architecture model over its peers by evaluating and analyzing the performance and this process is called as **Model Based Design Space Exploration.**

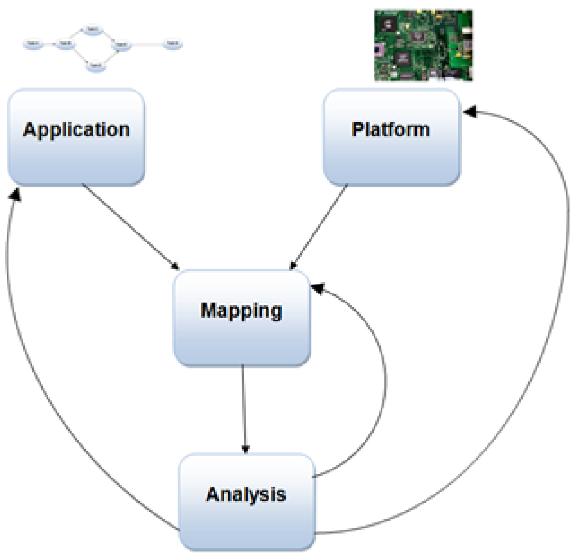


Figure 1.1: Y Chart Methodology

The figure above represents the Y chart methodology. This methodology was first proposed by [7]. As it is clear from the figure, the Y chart methodology has four important factors:

Application Layer : As the name indicates, this layer contains the set of applications and the behavior of these application can be obtained through the control dependency (dependence in execution of one task over the other task) to ensure equal workload across all processes and data dependency (interaction between the tasks) between the tasks and also the order in which they need to execute.

Platform Layer : The platform layer furnishes the characteristics of the platform to which the application will be mapped. These characteristics include the type of operating system being employed (Priority based, First Come First Served), resources like CPU, GPU, FPGA over which the tasks can run.

Mapping: This is the phase where the applications are mapped to the platform. After being mapped to the resources, the tasks in the applications are executed pertaining to the dependencies they have with each other.

Analysis: The performance of the mapped application is obtained and then it analyzed for bottlenecks in terms of speed, memory usage, power consumption, energy consumption etc. For any architecture, the performance is measured for various set of applications. These applications are mapped with a set of platform and then the performance will be analyzed. The designers then get a knowledge about the obtained performance, its bottlenecks, advantages and disadvantages and they pave the way for a new design which will have a different application or a platform or mapping respectively. This process is helpful to make design choices and can also be used as a big motivation for the selection of a particular choice respectively.

1.2 Data Path of Printer

The Data Path in a printer is a complete route of data from a source like a scanner or input PDF to the target, which is the print head. It plays a vital role in influencing the output quality. To explain its working in a nutshell, when a user provides an input for the printer by selecting the PDF file, the data path transforms this input file into matrices which in turn specify whether a nozzle present in a printer head must emit a drop or not. From [5], the schematic representation of a high level view of a data path on a high end copier is as follows:

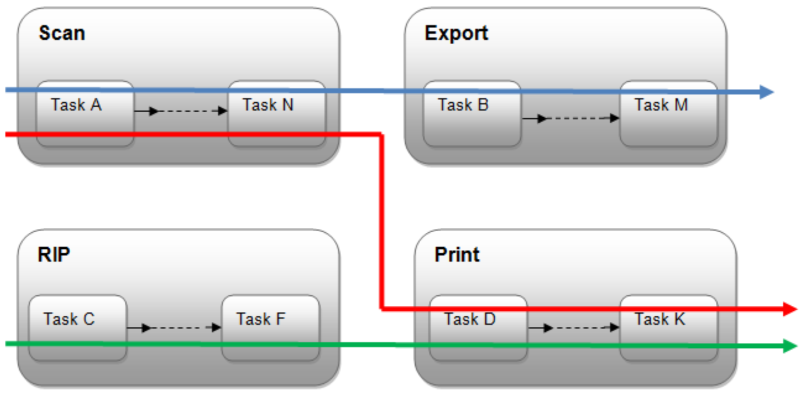


Figure 1.2: Data Path of a Printer

Considering the example of a PDF document being sent as an input by the user to be printed, the data path contains the image processing steps that are needed to be performed from the moment a PDF is being fed as an input to the moment the input reaches the print head.

The interesting factor here is that all these image processing are done real time. The Scan job in the above figure scans the image, then processes it and sends it over the network to their destination. The Print job basically receives an input file (such as PDF) and prints the necessary pages. The RIP expands for Raster Image processing and normally the print jobs are processed by the RIP and the Print path. The amount of data involved with this kind of image processing is huge as the jobs specified above has number of additional settings to take into consideration such as the type of the paper (A3, A4, etc.), the number of images for Copy and Print jobs, and the export format like (JPEG, PDF etc.) for Scan jobs respectively.

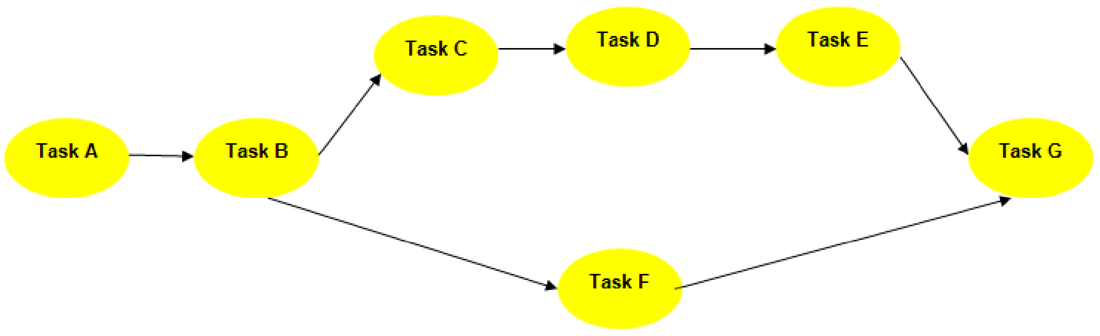


Figure 1.3: Task Graph of a Printer Data Path

The above figure is an task graph representation of a data path of a printer which was considered as an example system in this research. The precedent constraints are specified during the design phase. In the above case, an object of Task B can start only after the completion of the object of Task A. For example, if ten objects are given as an input to the system, A0 would start first and B0 can start only after the completion of A0. But, A1 and B0 can run parallely with each other and the same rule applies to all the remaining tasks in the system. With respect to any job like Scan, Export or Print, several processing operations/tasks such as scanner corrections, resampling, contrast enhancement needs to be adopted. The data handled for the operations are obviously high and hence in order to cope up with these data intensive behavior, they are sometimes implemented in hardware. Each task of a data path can be executed on a hardware platform such as CPU, GPU, and FPGA. The important factor to be noted here is the tradeoffs that need to be taken into account with respect to speed, power usage, cost, latency depending upon the requirement. Designing a data path like this is out of scope in this thesis project. But with these data intensive operations, the data path of a printer can easily be chosen as an example system for evaluating the performance of a library implemented using a modeling language.

1.3 Modeling Data Intensive Applications

As explained in previous section, modeling a system with large volumes of data is a difficult task. Therefore, a generic modeling blueprint for system level performance analysis has been proposed in [5]. The blueprint developed was abstract enough to be implemented in different modeling languages. The approach proposed was implemented in Java and UPPAAL in [5]. The blueprint was developed with an example system which mimics the behavior of a printer data path. Modeling the behavior of a data path case comes with several design challenges. For example, the task depend on a computational resource like CPU, GPU or FPGA and also a storage resource to progress in time. The speed of these tasks are set by the resources. There must be a claim and release mechanism, such that when a task is ready to run it claims and when it is finished, it releases the resource. Similarly, when two tasks run on a same computational resource platform like a CPU or a GPU, the utilization of the resource by the task must be shared. In other words, 100% utilization is no longer possible.

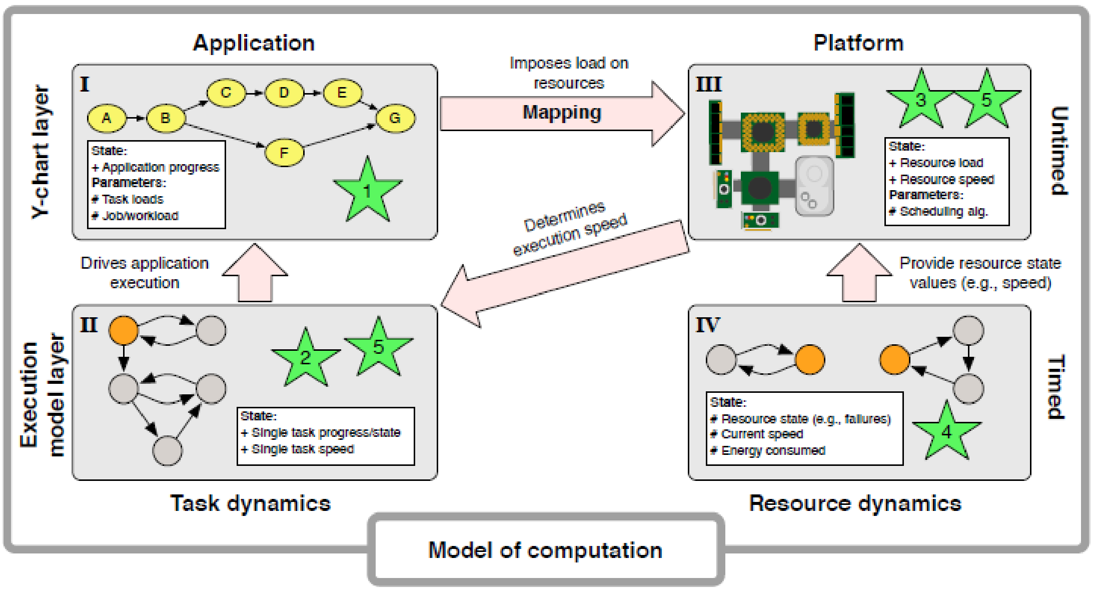


Figure 1.4: Blue Print Model from [5]

The above figure represents the modeling blueprint and the relationship between the different elements. It has two layers: Y chart layer and the Execution model layer.

The **Y chart layer** was already discussed in the previous section 1.3. The Application model contains the dependencies between the tasks in the form of a task graph and the platform contains the information about the resources to which the tasks will be mapped to. The resource information involves the type of resource (memory or computational resource) and the characteristics of these resources. But, the most interesting part here is the **Execution Model layer**.

The **Execution Model Layer** involves task and resource dynamics. At any time instant, the Task dynamics covers information whether a task is ready to run or not and also the information regarding whether a task is finished running or blocked during the course of the progress. The performance modeling in an application is carried out through following steps

- Modeling the Application
- Modeling the Platform
- Modeling the Task Dynamics
- Modeling the Resource execution behavior
- Evaluating the performance of the entire system

1.4 Overview

Chapter 2 describes the research questions and the definition of the problem that will be addressed and solved in this thesis project. *Chapter 3* gives a brief introduction on the software systems such as SystemC, SystemC-AMS and covers the evaluation of the performance criteria. It also describes the prior work in Java by [5] and small preparatory study that was performed before starting with the design of the system. *Chapter 4* explains the methodology in which Y chart Model was implemented. *Chapter 5* explains the implementation of State Manager in SystemC. *Chapter 6* addresses performance of different models (SystemC, C++ and Java) and the reason behind their performance. Finally, *Chapter 7* discusses the conclusion and some recommendations for the future.

Chapter 2

Problem Definition

Problem Description
Writing a simulation layer explicitly for a data intensive application is an extensive process. This thesis aims in solving this problem by using the simulator present in SystemC to simulate the behavior of the system. The implementation was then evaluated under different metrics to determine where it stands when being compared with other high level modeling languages such as Java and C++.

2.1 Research Questions

What did the previous Research focused upon?

This thesis extends the research of [5] on presenting a generic modeling blueprint for data intensive embedded application. The applicability of the approach has been tested in [5] through a framework which was simulated, developed and tested with a printer data path case using the tools UPPAAL and Java. The pros and cons of using Java and UPPAAL has also been addressed in [5]. Both modeling techniques have their own advantages and disadvantages. UPPAAL has higher support towards modeling the execution layer and also in analysis power, but it has problems with performance, scalability and integration. In case of Java, its fares better with performance, scalability and integration when being compared with the UPPAAL. But, the problem with Java implementation is that the simulation layer (which defines the behavior of the simulation) was written explicitly.

Is SystemC an interesting candidate compared to Java and UPPAAL?

SystemC is becoming an emerging language in the field of modeling both hardware and software components. It uses an open C++ library for system design and validation. SystemC has an in built simulator which could be used to perform simulations of data intensive applications like the data path of a printer case. The dynamics of SystemC and the working of its simulator is explained in detail in chapter 3.

The simulator in SystemC was used to carry out the entire simulation instead of explicitly writing the execution and simulation layer like Java or C++. In this research, the Y chartlayer of [5] was implemented in C++ and the execution layer was implemented in SystemC. This is explained in chapter 4 and chapter 5.

Any modelling language will have its own set of positive and negative aspects. In the same way, SystemC was evaluated under various criterias to determine its strength and weakness. This is explained in chapter 6.

What are the Modelling challenges in SystemC?

SystemC is a language developed by Open SystemC Initiative which is an open source environment. In this research, the modeling challenges were analyzed while designing the system and expressed through a learning curve in chapter 6 .

How does SystemC perform for a Data Intensive Application like a Printer data path case?

SystemC being a powerful modeling language was expected to perform well in terms of speed when compared with other High Level Modelling Languages such as C++ or Java.

As a part of this reseach, a library was developed and a data intensive application was simulated using the SystemC simulator and its performance was evaluated under different metrics in chapter 6.

Chapter 3

Domain Analysis

In order to address the research questions and solve the problem, an analysis must be carried out to study the properties of the software under consideration. This section covers the concept of discrete event simulation environment. In addition to that, the SystemC environment is explained and the dynamics of both SystemC and SystemC AMS are analyzed. As a last step, a simple system (proof of concept) has been built in both SystemC and SystemC AMS and the results are studied for its feasibility.

3.1 Discrete Event Simulation

In any Industry, simulation plays a pivotal role for its nature of allowing designers to analyze, design and test complex systems. The important advantage is that these experiments need not be physically carried out to know the results of the experimentation thereby saving a lot of expenditure. In addition to it, simulation can be used to predict the results for certain design decisions or the actions, the effect of modifications that are made and also it enhances the proper understanding of a system.

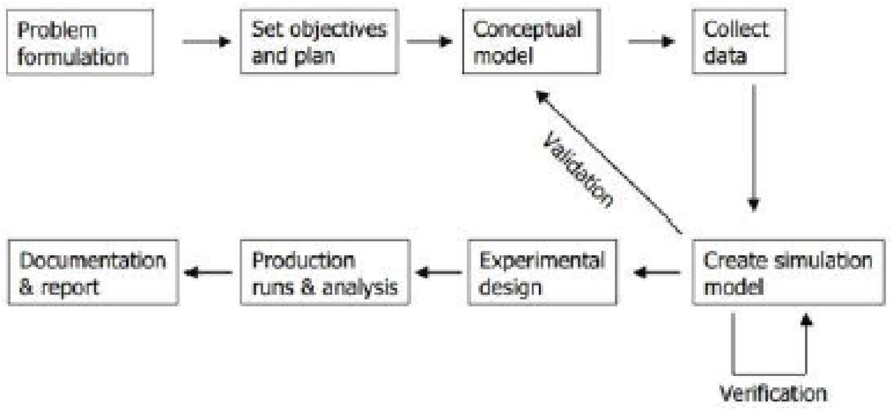


Figure 3.1: Discrete Event Simulation from [9]

The above figure represents the various stages of a simulation study. First step is to formulate a problem like what was discussed in the previous section. Then, the objective would be decided. A model will be developed in order to achieve the objective and then the simulation is run and tested. Once, the verification and validation of the simulation is performed, the design goes for production run and then documented for the future reference. The Simulation can be broadly classified into two types: Continuous Simulation and Discrete Event Simulation (DSE). Continuous Simulation is a process in which the behavior of a system will be characterized through a differential or differential algebraic equation. Thus, the time would be modeled as a continuous flow (i.e.) the behavior of the system changes gradually over time. For example, the position and velocity of an airplane flying in the air where its velocity and position changes continuously with respect to time. The use of Discrete Event Simulation or Continuous Simulation depends on the system under test.

The key feature of Discrete Event Simulation is its flexibility in modeling the system with various levels of complexity and detail, which makes it an interesting candidate in several industries. As stated in [9], another important advantage with it is the property of time handling. Discrete Event Simulation was first created during 1960s for analyzing and enhancing the research in industrial and the business process. It is a process where we analyze the performance of the models under study as a discrete sequence of events in time. There are different states associated with the system and a state change happens whenever an event occurs. The simulation is achieved by having a set of events to happen. When an event happens, the simulation time advances with respect to the event. In popular culture, the clients arriving in a bank can be considered as the suitable example. Here, the customer-arriving and customer-departing can be considered as events. In addition to this, the states of this system can be the number of persons in the queue and the service status provided for the customers. Thus the overall simulation time depends on the customers inter arrival time and the process service time provided by the bank. This simulation is basically a stochastic process.

The generic principle of discrete event simulation is as follows: when an event arrives, it gets added up in an event list/queue where the events are stored with time stamps and the event with smallest time stamp triggers the first execution and results in state change. Thus, the simulation time here can be advanced based on these events. In other words, the periods of inactivity are skipped over by jumping from event time to event time respectively. This is called as event driven Discrete Event Simulation.

3.2 SystemC

SystemC is a set of C++ classes and macros. It is used to describe a hardware/software model functionally, thereby implementing it on the final stage. Thus, the very basic question to be addressed here is the motivation behind using SystemC. The SystemC has a big advantage of creating an executable specification for the system to be developed. For example, in the problem considered, an example system of a data intensive printer application must be modeled. Using SystemC and its libraries Discrete Event Simulation environment can be created through the specifications [10]. In addition to this, the SystemC also provides a rich set of data types to model our system pertaining to the specifications. The SystemC design methodology can be understood in detail through the following figure.

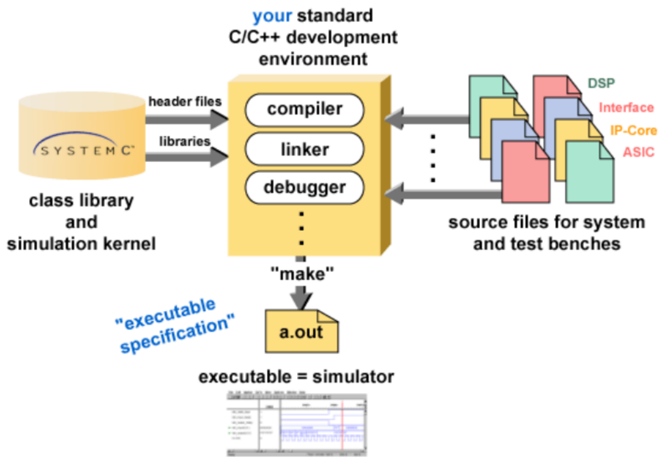


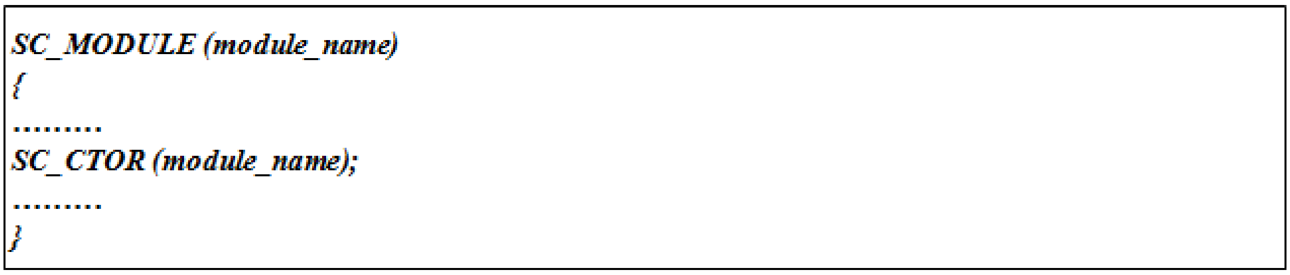
Figure 3.2: SystemC Compilation from [8]

The main advantage of using SystemC is that it is written in only one language and translation is not necessary. As explained in the above figure, a designer can write a model using the C++ language with additional functionalities offered by the SystemC standard. Thus, the model written is an executable specification and can be compiled and linked with the SystemC library and the simulation kernel. The three major building blocks of the SystemC are:

- Module.
- Port and Signal.
- Process and Thread.

3.2.1 Module

The first building block of SystemC is the Module. It is mainly used to break a complex design into different simple parts. The main advantage of using it is to hide the information and data from other modules. The modules have an inbuilt macro and can be declared as follows:



```
SC_MODULE (module_name)
{
.........
SC_CTOR (module_name);
.........
}
```

Figure 3.3: Syntax of a SystemC Module

3.2.2 Port and Signal

Every module interacts with each other using ports. In other words, each module can have input and output ports. The communication they establish can be explained through the following example Both the modules have two ports, an input port and an output port. These

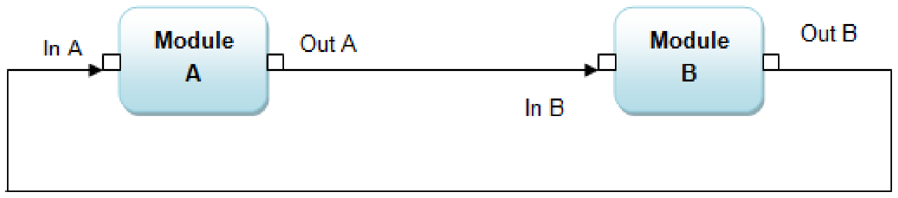


Figure 3.4: Example System With Ports and Signals

ports are normally connected in order to establish communication between the modules. The connection is established through signals which can be declared in the main function by using ***sc_signal*** and it can be used to connect the instantiation of ports from individual modules. In the example figure 3.4 the InA, InB are the input ports and outA, outB are the output ports. These ports are defined through following syntax

sc_in< T >Port_Name

The sc_in is the keyword for the input port declaration; T refers a generic data type as the ports can be of data type like boolean, integer etc. and then the port is given a name. The same syntax applies for output port as follows:

sc_out< T >Port_Name

In addition to it, each module has sensitivity to an input port or an event which means that the processes in the module would be activated whenever an input is received through these input ports or a notification of an event to which the module is sensitive to. This can be specified explicitly as following using the *sensitive* keyword

sensitive <<inA
sensitive <<event

3.2.3 Process

In a SystemC module, there are two types of processes: One is called ***Method*** and other is called ***Thread***. In the case of Method, it is defined with the help of the macro **SC_METHOD** and the important factor here is, once this method is called during simulation, it can never be interrupted in between. In other words, there are no infinite loops allowed here and also the process cannot use wait() statements in-between the execution.

The Thread execution is opposite to that of the SystemC method. It is defined with the help of macro SC_THREAD and in this case the module has its own thread of execution and can be interrupted or made to wait using wait() statements and starts again after waiting for a particular event or a period of time. The main difference between a SystemC thread and a real time thread is that, the SystemC thread uses the concept of ***Co-operative Scheduling***. In other words, if there are two threads A and B which can run parallel, then A will run till it encounters a wait() statement. Once it encounters an wait() statement, the scheduler is passed over to B and the thread B will run till it encounters the first wait () statement and this repeats till the simulation is finished.

3.2.4 Wait Statements in SystemC Threads

The wait statements could be expressed in different formats in SystemC with each having a specific functionality of its own.

wait()

```
SC_MODULE(waitTest)
{
int i=0;
void run()
{
while(true)
{
ev1.notify(1,SC_NS);
ev2.notify(4,SC_NS);
wait(ev1 | ev2);
}
}
SC_CTOR(waitTest)
{
SC_THREAD(run);
sensitive << ev1;
sensitive << ev2;
}
};
```

Listing 3.1: Example of Wait Statement

SystemC threads can be sensitive to one or more events. The wait statement without any parameter makes the SystemC thread to wait for any of the event in sensitivity list to occur. In other words, it waits for the earliest event of all the events to which the SystemC thread is sensitive to.

In the above example, though the thread is sentive to both the events ev1 and ev2. The ev1 advances time for 1 nano seconds (SC_NS) and ev2 advances the time for 4 nano seonds. The important thing to be noted here is that for each step the time for the thread will be advanced only by 1 nanosecond as the wait statement suspends the thread only for the notification from ev1 as it is the least one and it happens first.

wait(event)

The above statement makes the thread to wait till the event mentioned as parameter occurs. If the corresponding event is notified for a time period, then the thread resumes with the updated time stamp.

wait(event1 |event2)

The SystemC thread waits for any one of the event given as the parameter to occur.

wait(Time_Period,Time_Unit)

When a thread encounters a wait statement like above, then it will wait till the time period specified to be activated again.

wait(Time_Period,Time_Unit,event)

The above statement makes the thread wait for a specific time period or for a particular event, whichever occurs first.

Data Types

SystemC supports all C++ data types such as int, long, short, double, bool, unsigned int etc. In addition to this, it also provides SystemC data types for hardware modeling respectively. One type of the SystemC data type is the *sc_bit*. It is the data type which takes values 0 and 1. This is used when there is no need of Z(hi impedance) or X(unknown) values. There are number of operators such as bitwise AND, bitwise OR, bitwise XOR, AND assignment, OR assignment etc. But, when real time hardware is modeled, X and Z would be used and in that case SystemC provides *sc_logic*. The interesting feature is that *sc_bit* and *sc_logic* can be assigned to each other. Another interesting data type offered by SystemC is fixed and arbitrary precision which allows the designer to fix a precision (like 16 bit or 32 bit) if there is a prior knowledge about the values a parameter would take. The *sc_int* and *sc_unit* can be used to model data up to 64 bits.

3.2.5 Simulation in SystemC

The below figure 3.5 represents the simulation process of a SystemC scheduler. As the first step, the process/threads will be initialized. Then, the process(i.e) the SystemC methods will be executed fully or the threads will be executed till its first synchronization point. As the next step, the evaluation phase starts. In this phase, the events which are notified immediately will run the processes in the same phase. As it is clear from the figure, the process will be in this phase till there is no more process to run. Then, the values of the channels of the previous evaluation phase will be updated. This continues till there is no more event notification signaling the termination of the simulation. Then it moves to next simulation which has pending events and starts back from step2 again. Each phase of SystemC scheduler is illustrated in detail as follows

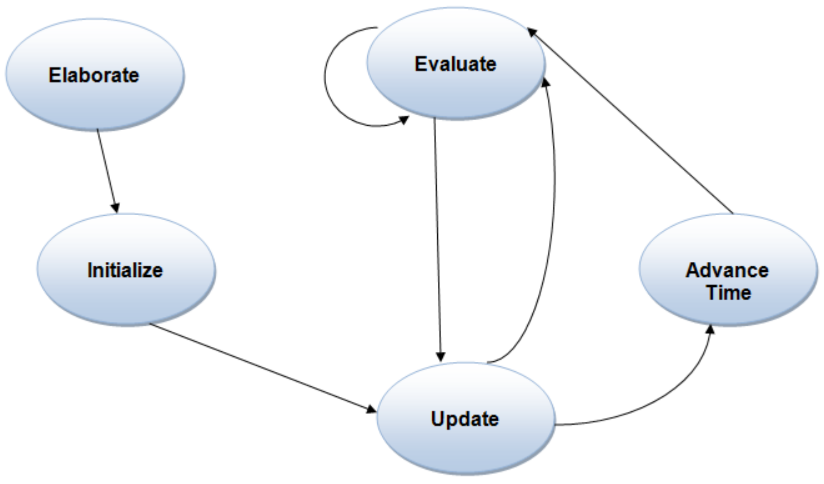


Figure 3.5: SystemC Simulation Steps

Elaboration Phase

This is the first phase in SystemC simulation. The different modules which are associated with the design will be instantiated here. It will be elaborated with all the parameters it needs to be constructed with. As discussed in previous section, several modules will be interacting with each other and the signals which connect the ports of these modules will also be elaborated here and the connection would be established. For example, let us consider two modules M1, M2 and the signal SIG which connects the output of module M1 to input of module M2 in order to establish the communication between the two.

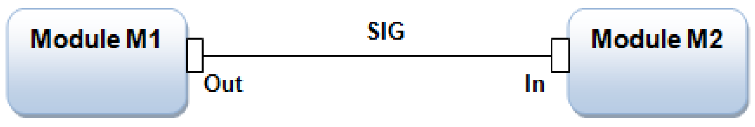


Figure 3.6: Communication between Modules

The Elaboration for the above example model will be written in SystemC as follows:

```
sc_main()
{
Module M1(...); \\M1 is the object of Module 1
Module M2(...); \\M2 is the object of Module 2

sc_signal<T> SIG;

M1.out(SIG) \\ out is the output port of Module M1
M1.in(SIG) \\in is the input port of Module M2
......
}
```

Listing 3.2: Elaboration Phase

As it is clear from the above fragment, the modules are instantiated , then the signal is defined and then the communication is established between the modules using the signal SIG.

Initialization Phase

The next phase in the simulation is the initialization phase. The start of the simulation is initialized here with the ***sc_start()*** as we can see from the below figure. Practically, the simulation will never start unless this keyword is specified explicitly.

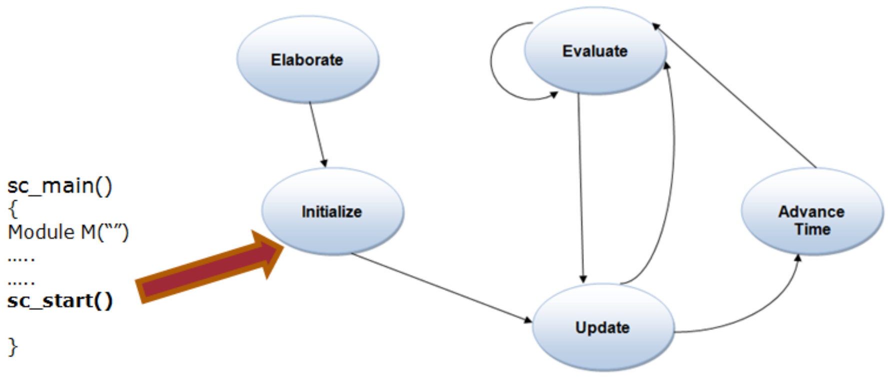


Figure 3.7: Initialization Phase

Evaluation and Update Phase

Once the simulation starts, the progress of each and every process heavily depends on the evaluation and the updating phase. From [5], the simulation in SystemC can be summed up as following

1. The simulation starts immediately after the initialization phase with a set of runnable process.
2. All the processes (both SC_THREAD or SC_METHOD) are runnable at the start of simulation except in the case if a process has a dont_initialize() function call.
3. Each runnable process is executed in the evaluation Phase in any order till it encounters a wait() statement(if it is a SC_THREAD) or finishes its execution completely(if it is a SC_METHOD).
4. If there are no more runnable process, then the update Phase comes in to action updating all the SystemC variables and also executing the event notifications, thereby advancing the time if needed. After update phase, all the process which are sensitive to the notified event will be added to runnable process for evaluation and the process would be repeated again.

5. The **delta cycle** in SystemC is the transition from the evaluation phase to the update phase and back to evaluation phase respectively.
6. In SystemC, we have three types of event notifications. They are Timed, Untimed, Zero Time.

3.2.6 Event Notification

Timed

event.notify(2, SC_NS) : Here, an event is notified and will advance the time by 2 nanoseconds (the time could also be milliseconds, microseconds, picoseconds). Any process which is sensitive to this event will be updated with an advance in time of 2 nanoseconds. If this statement is encountered during the evaluation phase, then the event will be notified during update phase and it will eventually advance the time.

Immediate

event.notify(): This kind of notification is completely different from the above notification. If this statement is encountered in the evaluation phase, then the event will be notified immediately there by adding all the processes which are sensitive to the event to the set of runnable process.

Zero Time

event.notify(SC_ZERO_TIME): If an event is notified with a SC_ZERO_TIME as parameter, then any process which is waiting for this event will be triggered after one delta cycle (i.e.) after an update phase. But, the time will not be advanced though.

The SystemC simulation engines behavior can be understood by the following example

```
SC_MODULE(TestSystem)
{
sc_out<bool> outport;

void process1()
{
int j = 5;
ev1.notify();
wait(ev2);
int k = j + 2;
}

void process2()
{
outport.write(true);
ev2.notify(3, SC_NS);
cout << "Process 2" << endl;
}

void process3()
{
wait(ev1 | ev2);
int l = 0;
}

SC_CTOR(TestSystem)
{
SC_THREAD(process1);
SC_THREAD(process2);
SC_THREAD(process3);
sensitive << ev1;
sensitive << ev2;
}
};
```

Listing 3.3: SystemC Description

The example above consists of 3 processes which are SystemC threads and they can start in any order. Considering the scenario Process 1 starts first, followed by process 2 and then process 3, the following behavior will be exhibited by the above module:

- The **evaluation** process will start with process1(). The variable j will be initialized to the value. The event gets notified for immediate notification, hence the process (in this case, process 3()) which is sensitive to the event gets added to the set of runnable Process. Now Process 1 will be terminated as it encounters the wait statement thereby giving the simulator for the other processes which are ready to run.

- Next Process 2() will start, the output port will be written a value of true. But, the important point to be noted here is unlike the variable j, the outport will be updated with a value of true only in **update phase** (i.e.) when there is no more runnable process. The event ev2 is said to advance time for 2 nanoseconds, which will also be done in the update phase. This process now gets suspended as it encounters the wait statement.

- Process 3() starts. The wait (ev1 |ev2) means that the process waits for which ever event happens first. As ev1 is already notified immediately by process 1(), it will happen first than ev2, hence process3() would run till it encounters an wait statement

- As there are no more processes which are ready to run, the update phase starts thereby the value of *outport* would be updated and notifying the event. Now, the runnable processes are checked and the evaluation phase starts thereby repeating the above process till there are no more processes to run.

3.3 SystemC-AMS

SystemC AMS stands for Analog and Mixed Signal Design. It is the extension of the existing SystemC language. It was downloaded as SystemC AMS 2.0 beta version which was released by Fraunhofer-Gesellschaft [3]. This is the latest version being released and it was used to build a proof of concept which is discussed in the next section. As the name suggests, the main objective of this is to develop analog and mixed signal systems by extending the SystemC language. In other words, the SystemC type of methodology can be used to develop, refine and verify the mixed and analog systems respectively [1].

The figure 3.8 represents the extensibility offered by the SystemC AMS. By using SystemC kernel, it allows three different modeling formalisms: Timed Data Flow (TDF), Linear Signal Flow (LSF) and Electrical Linear networks (ELN)[4]. Out of these three formalisms, only the TDF model of computation was studied as it was relevant to the topic of discussion. Hence, the language of SystemC AMS and the TDF formalism is discussed briefly in the following sections:

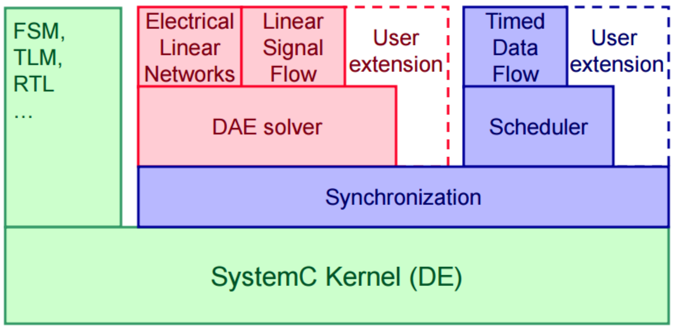


Figure 3.8: SystemC-AMS from [4]

3.3.1 SystemC-AMS Language

The SystemC- AMS language is very much similar to that of SystemC. Only the keywords used are different. For example, the modules are defined in the SystemC AMS with **SCA_MODULE()** [9]. The important thing to be noted here is that, the modeling formalism being used must be explicitly specified. For example, in our case the TDF module can be defined by using the macro **SCA_TDF_MODULE()**. Like module definitions, the ports definitions are also similar to that of SystemC, where ***sca_in<T>*** and ***sca_out<T>*** was used for input and output ports respectively. The modules of AMS could be wrapped inside a SystemC module. The signal in SystemC-AMS was created using the syntax ***sca_signal<T>*** which is used to connect two SystemC AMS ports and a normal SystemC signal could also be used if discrete event converter ports are used in a module. Thus, the compilation of an AMS model is similar to that of the SystemC as discussed before except for the fact that, we have a SystemC AMS library being included to invoke the functionalities.

3.3.2 Timed Data Flow (TDF) Module

The execution of the module is similar to that of the SystemC module [9], where the module reads the input ports, processes the calculations and writes back the value to the output port. In this case, the module is said to be a continuous TDF module and the properties of a continuous TDF module can be summarized as follows:

- Each module execution is tagged with an absolute time point by the Timed Data Flow.

- Hence, the time step/the time difference between the two module executions are always a constant.
- The time distance/ time step must be specified explicitly in a Timed Data Flow module.
- It also enables to synchronize with the time driven simulation like SystemC MoC and embedding of time dependent functions like continuous transfer function.

3.3.3 Loop in Timed Data Flow

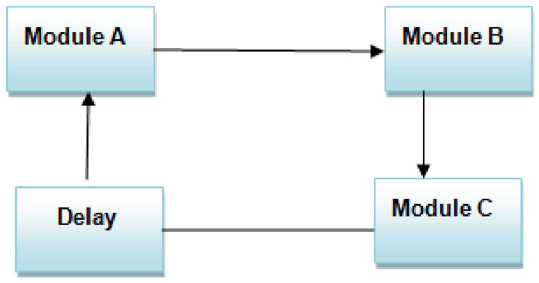


Figure 3.9: Loop Scenario in SystemC-AMS

As shown in figure, loops in TDF must be handled carefully. If not, then it would result in cyclic dependency. This scenario can be averted by introducing a delay in the firing of the next sample. Thus, a module without a delay would not be schedulable. But, it is also possible to connect a TDF model with a discrete event domain using the TDF converter ports as follows:

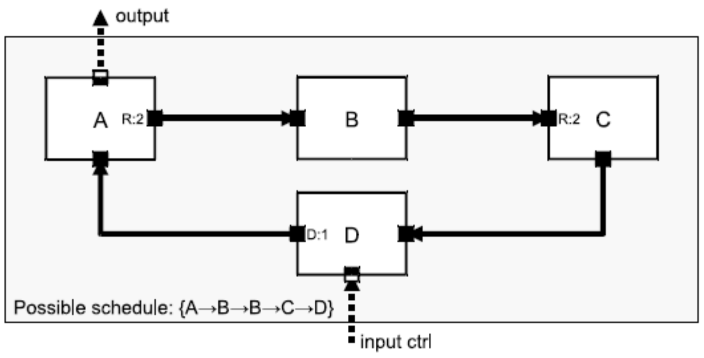


Figure 3.10: Discrete Event Convereter Ports from [6]

In the above case, the module D has a converter input port which reads a discrete event signal and the module A has a TDF converter port which outputs a discrete event signal. Here, the discrete event converter ports are given as dashed arrows. [6] When using a discrete event signal interaction, special care must be taken. In other words, SystemC signals must be used in order to make the modules communicate effectively. A TDF could become a part of closed loop having a path through discrete event domain as follows:

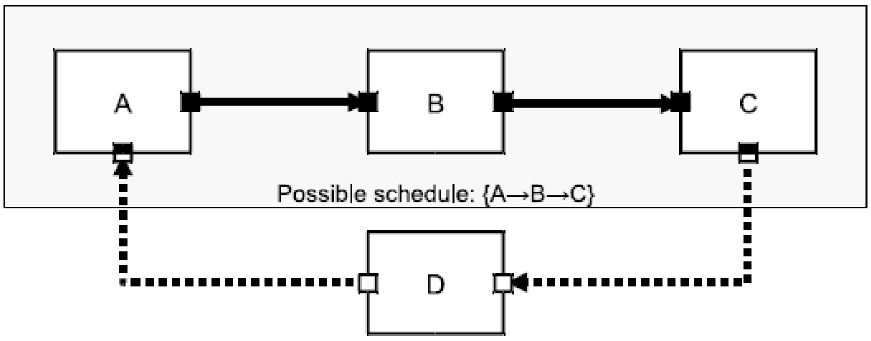
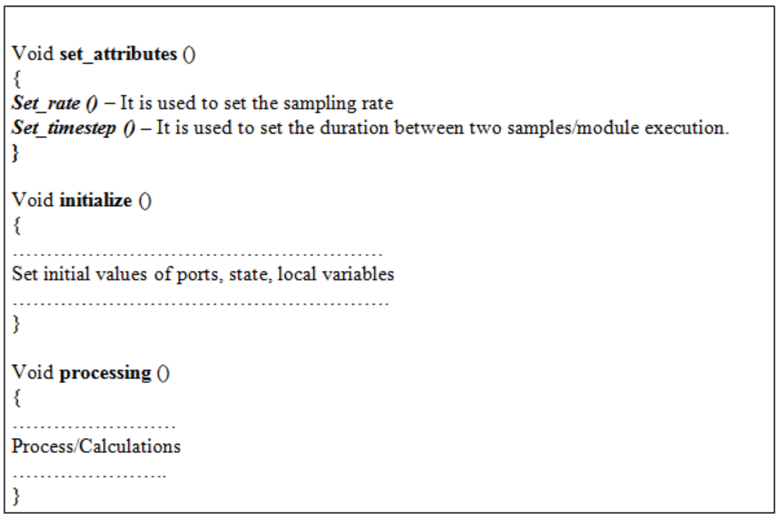


Figure 3.11: Loop with Discrete Event Ports from [6]

Here in the above case TDF module contains no loop and hence no delay is needed for scheduling. [6] In other words, the module A reads a sample from the discrete event domain at first delta cycle associated with a sample using a TDF converter port and the module C writes a sample to the discrete event domain in the same delta cycle.

3.3.4 TDF Declarations

In order to execute a TDF module, we need the following attributes to be explicitly specified:

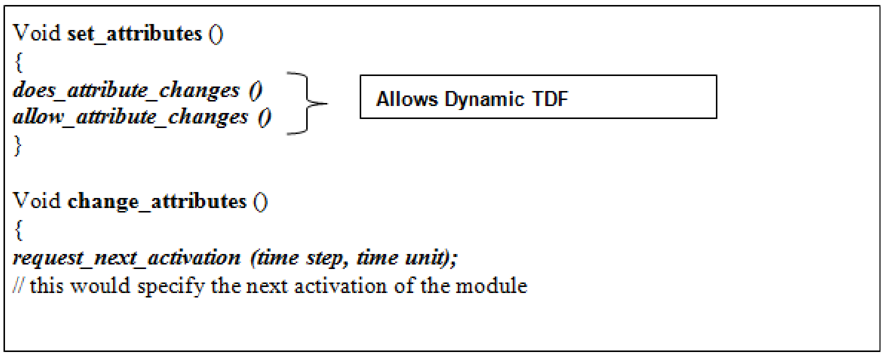


```
Void set_attributes ()
{
Set_rate () – It is used to set the sampling rate
Set_timestep () – It is used to set the duration between two samples/module execution.
}

Void initialize ()
{
.....................................................
Set initial values of ports, state, local variables
.....................................................
}

Void processing ()
{
........................
Process/Calculations
........................
}
```

Figure 3.12: Sytax of a TDF Module

3.3.5 Dynamic TDF

Till SystemC AMS 1.0 version, only continuous TDF was used. The computations would be executed in discrete time steps considering the input samples as the continuous time signal. In other words, the continuous TDF executes for every clock tick. Contrary to that, the dynamic TDF doesnt execute at every clock tick. Rather, it would not activate or calculate any values until it is been requested by other module. This dynamic feature of TDF can be used for Discrete Event Simulation. There is a separate function called **change_attributes()** which can be used to change the value of the time step depending upon the requirement and we need to use **request_next_activation()**. Thus, the declaration of Dynamic TDF in a module can be written as follows:



```
Void set_attributes ()
{
does_attribute_changes ()
allow_attribute_changes ()     Allows Dynamic TDF
}

Void change_attributes ()
{
request_next_activation (time step, time unit);
// this would specify the next activation of the module
```

Figure 3.13: Sytax of a Dynamic TDF Module

3.4 Java Implementation

As discussed in the problem description section, a generic example mimicing the data path of a printer was already implemented in the Java programming language. This section of the report aims at briefly describing the principle behind the implementation and also the simulation results respectively. The example system [5] which was considered for the implementation is as follows:

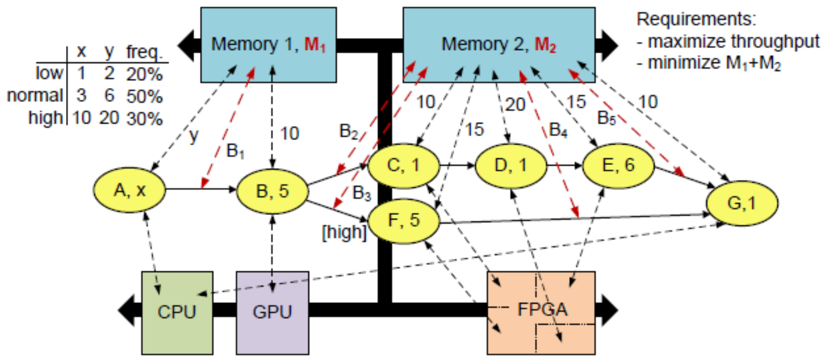


Figure 3.14: Example System from [5]

The application consists of seven tasks. The tasks are mapped to the resources like CPU, GPU, FPGA, and Memory. The tasks A and G are mapped to the CPU, task B in GPU and the rest of the tasks in FPGA. The arrows between the tasks describe the precedence constraints. The numbers next to the task names are the loads associated with the tasks which are imposed on the memory and the resources respectively. If two tasks share a same resource, then they are given 50% utilization by the resource till one of the tasks completes.

For example, the progress abstraction of three objects sharing a same resource can be given as follows

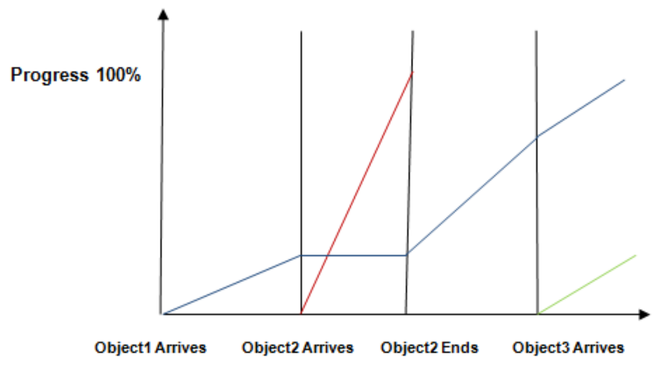


Figure 3.15: Progress Abstraction

Here, object 1 arrives first and it runs with the full speed till object 3 arrives, then both object 1 and object 3 can be given 50% utilization. This scheduling algorithm represnts the task interactions on very high level.This kind of task interaction was also achieved using SystemC and the steps in which it was achieved are explained in chapter 5.

3.4.1 Task Dynamics

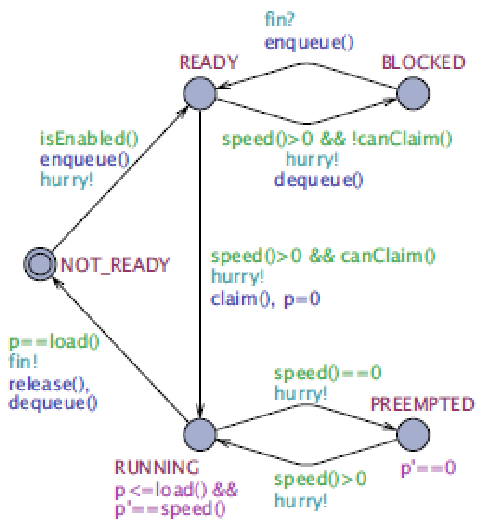


Figure 3.16: Task Automaton from [5]

The above automaton was developed in UPPAAL as a part of [5] to describe the different states of the task and the actions it takes to move from one state to another. The task has five states and it moves from one state to another if it satisfoes the guard condition. The five states are defined as follows:

NOT_READY: This is the state where not all the inputs are available and hence it is not functionally enabled.
READY: This process is enabled on the functional level, but it doesnt have any progress because of the lack of available resources.
BLOCKED: The task has a non zero fraction of primary resource, but it cannot claim its storage resources.
RUNNING: As the name suggests, the tasks has available storage resources and it makes progress.
PREEMPTED: Thus, in this state a task which had its primary resource has been stopped from making progress due to arrival of a higher priority task. Though it cannot make progress for the moment, it holds on to its resources.

The transition happens from one state of the above to other through enqueue(), dequeue(), claim() and release(). [5]The enqueue and dequeue pertain to the primary resources of the task and it adds or removes the tasks to the list of the tasks that need to use the primary resource. The claim and release are normally used for the storage resources. Thus, all the blocked tasks are unblocked once the resources become available to it. Then it moves from the RUNNING to READY state respectively.

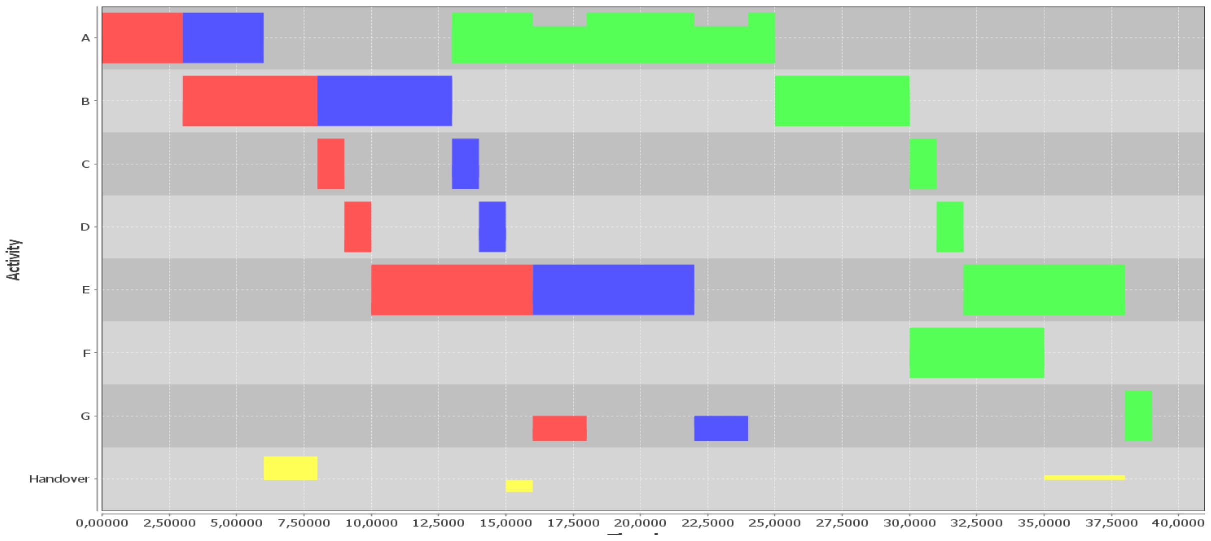


Figure 3.17: Output Trace Obtained from Java Implementation

The above figure is the trace obtained for the system discussed above having 3 objects of input respectively. Each block here represnts the object of a respective task. The scheduling of two tasks sharing a same resource can be visualized here. Consider Task A (Green) and Task G (Red). When the Task G arrives, Task A and Task G is given 50% utilization each and then once the Task G finishes, the Task A is given full CPU again.

3.5 Proof of Concept

As a part of preparation, a scheduling algorithm was implemented in SystemC, SystemC AMS and java in order to evaluate the performance of each modeling technique. The algorithm was also implemented in C++ coding which uses the SystemC clock for the scheduling respectively. As a whole, five implementations were done, which are as follows:

1. SystemC Implementation using Ports and Signals.
2. SystemC AMS Implementation using Ports and Signals.
3. SystemC Implementation without using Ports and Signals.
4. SystemC AMS Implementation without using Ports and Signals.
5. Java Implementation.

3.5.1 Scheduling Algorithm

The main aim of deciding on this scheduling algorithm is to mimic the behavior of the implementation done above. It works in the principle that if only one task is present at any time instant, then it is served fully by the resource (CPU in this case) and if there are two tasks, then both the tasks are given half utilization (or) 50% utilization respectively. For example, if Task A arrives at time 0ns and there are no more Tasks competing for the resource, then it is served fully by the CPU.

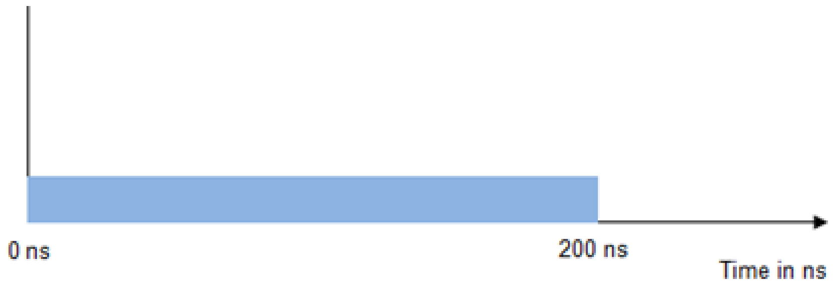


Figure 3.18: Task A progress Graph

In the above case, TaskA has 10,000 cycles and CPU has a processing of 50 cycles/Time period. Thus, as there are no tasks, the Task A is expected to finish by 200ns. But, If a Task B arrives at 20ns, with a capacity of 5000 cycles, then CPU provides 50% utilization for both the tasks till one of the task completes (Task B in this case) and gives back 100% utilization again to the task which needs to be finished with some remaining cycles. It is illustrated in the below figure.

The Blue represent the full utilization of the CPU by Task A and the red represents the 50% utilization of both Task A and Task B till the end of Task B. This behavior was adopted in all the implementations and the principle behind each implementation is given in the following sections.

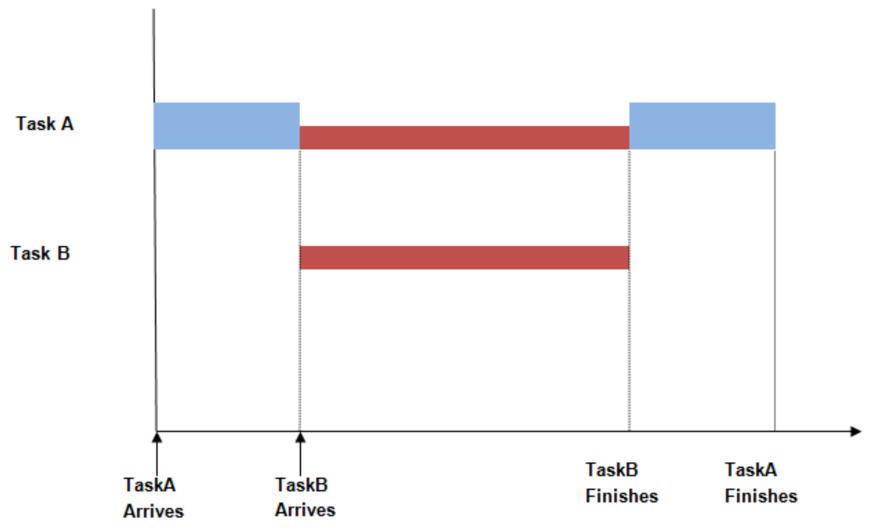


Figure 3.19: Parallel Execution of Tasks

Table 3.1: Simulation Performance Results

S.No	Implemntation	Execution Time
1	SystemC with Ports and Signals	1.357
2	SystemC AMS with Ports and Signals	27.12
3	SystemC without Ports	0.405
4	SystemC AMS without Ports	11.60
5	Java	0.219

The Simulation was performed for 100,000 simulation runs. In other words, the scheduling algorithm was repeated for 100,000 times and the above computation time values were recorded. The Implementation of SystemC, SystemC AMS with and without signals was measured in Microsoft Visual Studio 13.0 and the Java implementation was measured in the Java IDE Eclipse. From the table, it is clear that the SystemC implementation with the ports and also the SystemC implementation without ports fairs better than Java implementation. Hence, these two implementations were considered for the modeling of the example system which is already implemented in Java. The SystemC AMS is said to be the slowest of the lot and hence it is not considered for modeling the example system.

The important thing to me noted here is that the implementation in SystemC was done with SystemC methods instead of SystemC threads and also the ports played an important role in communicating the completion of tasks. But, the implementation of the bigger system (i.e. the example system explained in the problem statement) was implemented in SystemC threads. The design of the system is fully furnished in the chapters 4 and 5.

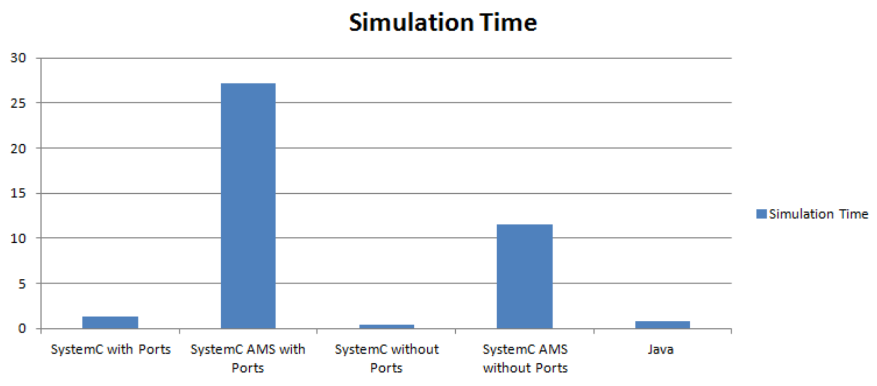


Figure 3.20: Results in Graphical Representation

3.6 Selection of Evaluation Criteria

For any implementation, evaluation plays a very important role. It helps in assessing the implementation and finding out its strong and weak points. In this research, the implementation performed in SystemC and C++ was evaluated across four different criterias. One is to assess the speed of the simulation which is the important objective of this research. In addition to it, the effect of increasing the size of the system was also analyzed to characterize the change in simulation speed. The challenges faced during the implementation of the system using SystemC was also analyzed as a part of this research.

3.6.1 Simulation Time

The simulation time can correspond to two things: One is the time which is taken to run a complete simulation of the model, where the time basically refers to the time recorded by the clock of the system (in this case the PC) in which the model is run. The other kind of simulation time refers to the time taken by the clock of modeling language. In this evaluation, the execution time (i.e.) the time for the complete simulation of the model is only measured and it is measure using the system clock of Intel®Core ^{TM} i5-4570 CPU 3.20 GHz. In this project, the simulation time was recorded as follows: the model would be implemented in SystemC language for different inputs and different systems. Then, the model would be made to run till it finishes its simulation and the total time taken to finish would be measured and evaluated.

3.6.2 Scalability

The scalability is the ability of the model to perform well or carry on with its functionalities when the size of the system increases. The data path example system has a total of seven tasks. In addition to it, two more systems were developed. One was a system with two tasks and another was a system with five tasks. The performance of each system was tested in order to analyze the speed of the implementation as the number of tasks in the system increases.

3.6.3 Program Complexity

The documentation with respect to SystemC for modeling is relatively less making it a challenging process to develop a generic library for simulation. In this project, the ease of modeling would be discused in order to throw light on the problems that were encountered while implementing with SystemC. Also, the code size of the execution layer implemented using SystemC would be compared with the simulation layer written explicitly with Java in order to obtain a picture of the gap that exists between the two above mentioned implementations.

3.6.4 Memory Usage

Most of the programms would get loaded on the memory and while they are running, additional memory gets allocated to store program level data. As SystemC is a set of C++ macros and a part of the design was also built using C++, the allocation and deallocation of objects play a crucial role. For increased loads, the increase in memory usage would be studied in order to understand the behaviour of the system and also its stability.

Chapter 4

Y Chart Layer Implementation in C++

The system as a whole is divided into two parts : Y chart layer and the execution layer. The Y chart layer is responsible for determining the behaviour of the tasks and resources and the mapping between them. The execution layer creates the task automaton and simulation will be carried out by performing the transitions and actions of each individual task through a state manager. In [5], the simulation layer was written explicitly in Java to perform these transitions and actions.

This approach of writing the simulation layer was avoided in this reasearch by using SystemC scheduler having the State Manager as the SystemC module. The reason for choosing the State Manager is explained in the section 5.2. The design process also involved the use of SystemC ports in order to study the effect of communication overhead. The following figure provides the overview of the implementation.

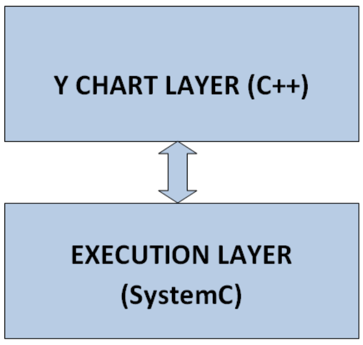


Figure 4.1: Overview of the Implementation

To sum up, the following implementations were made for each system that was designed as a part of this research

1. SystemC Implementation Using Ports

2. SystemC Implementation Without Ports

3. A Pure C++ Implementation

This chapter explains the design procedure of Y chart. As described in chapter 1, the Y chart layer has two layers: an Application layer and the Platform layer. This section covers the design procedure undertook for its implementation in C++.

4.1 Y Chart Layer : Application

The Application layer consists of the tasks and their properties. These properties include load, name, speed of the tasks, the storage it needs, the amount of resource it needs etc. An interface Y Chart Entity was created and the operations were made to be inherited by both the application (task) and the platform (resource).

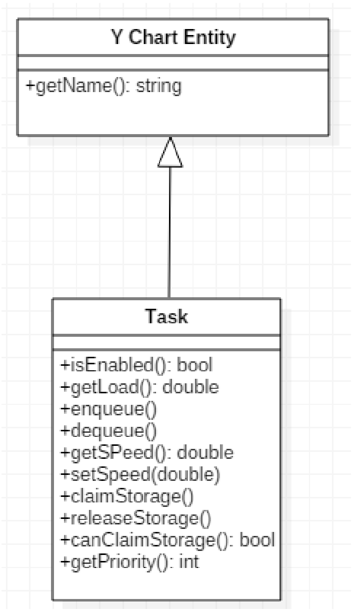


Figure 4.2: Task Interface

The operations for a task could be observed from the figure 4.2. An important thing to be noted here is that the above Task class was created as an Interface. The implementation was taken care by defining two more classes: *AbstractTask* and the *ObjectAwareTask*.

The **AbstractTask** is the class which basically gets constructed with the name of the task and the computation resource (such as CPU, GPU or FPGA) to which it is going to be mapped with. In addition to that, the methods to enqueue a task over a computational resource and dequeue the task after the completion is defined here. As computational resources decide the speed in which a task would run during the simulation, the speed of a task would be set in this class through the setSpeed() method and the speed of any task during run time could be obtained from the getSpeed() method defined in this class. In a nutshell, the interaction between the tasks and the computational resources would be defined here.

Every task in the system directly inherits from **ObjectAwareTask**. It gets constructed with a task name, the storage resources a task uses and the priority of each task. There are two storage resources used in this system. Another important thing to be noted here is that the ObjectAwareTask class inherits from the AbstractTask class. As the storage resources also play a pivotal role in the progress of a task over the course of time, the methods for claiming a resource and releasing it back was defined here. Every task gets an object type as its input which could be HIGH, LOW or NORMAL depending upon the load. Thus, the methods to add the data to a task (add()), removing data from the task (remove()) and also the current data(cur()) which is being used as input could be obtained from the ObjectAwareTask class. Thus, the complete class structure of the Application layer (Task) could be summed up with the following class diagram:

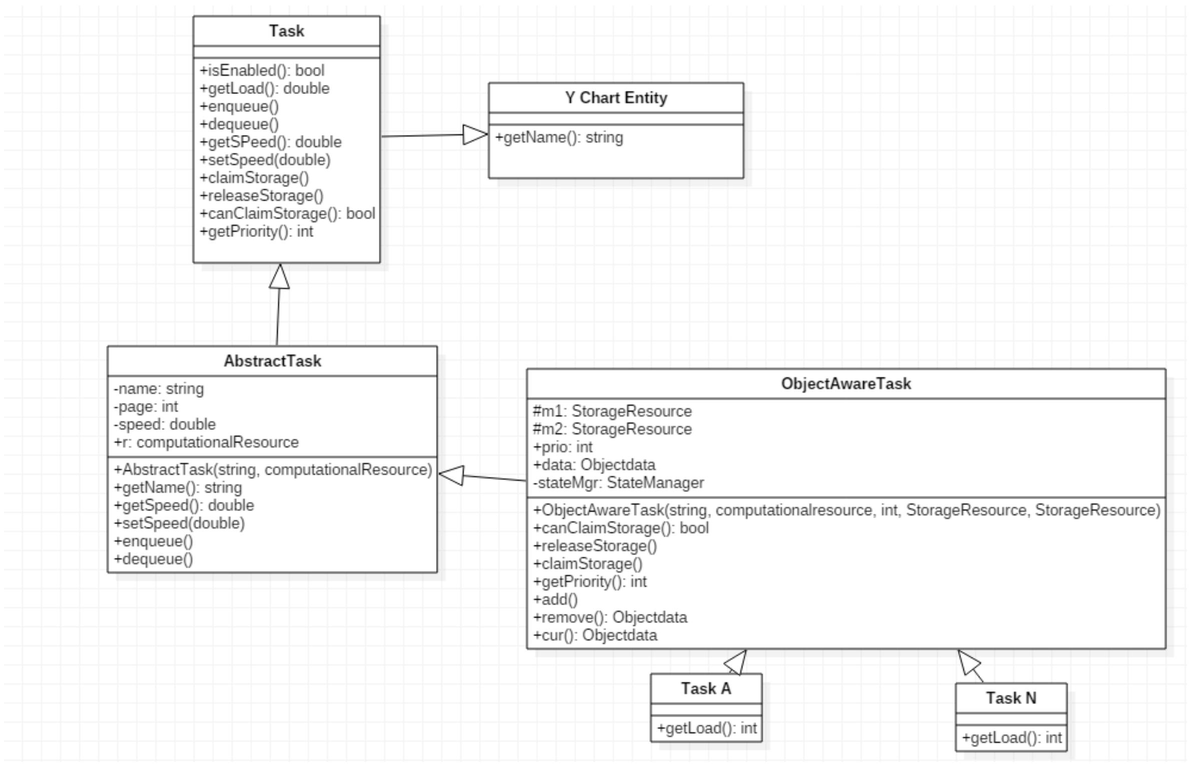


Figure 4.3: Application Layer

Each task in an application is created by inheriting the ObjectAwareTask class and it would be created with all important parameters. The load of each task and the amount of storage resource it needs to claim would be defined within respective task classes.

4.2 Y Chart Layer : Platform

As described in chapter 1, the platform layer is the layer which furnishes the information of the resource to which the application is going to mapped. In this case, two resources will be needed by the application to progress. They are the computational resources like CPU, GPU, FPGA etc. and the storage resource such as memory respectively.

The class diagram 4.4 represents the interface created for the platform layer. There is a parent resource interface which will be inherited by the interface for the computational resource and the storage resource. The tasks will enqueue on a computation resource when it is ready to run and it would dequeue from a resource when it has finished running. Thus, the speed of the task is determined by how many tasks enqueue upon a specific resource. If only one task gets mapped to a resource, then it gets 100% utilization (i.e.) it runs in full speed. But, if there are two tasks, then the speed for be equally divided for both the tasks there by giving each task only 50% utilization (i.e.) half the speed, until any one of the task completes its execution thereby giving the full utilization for the other task which is still running

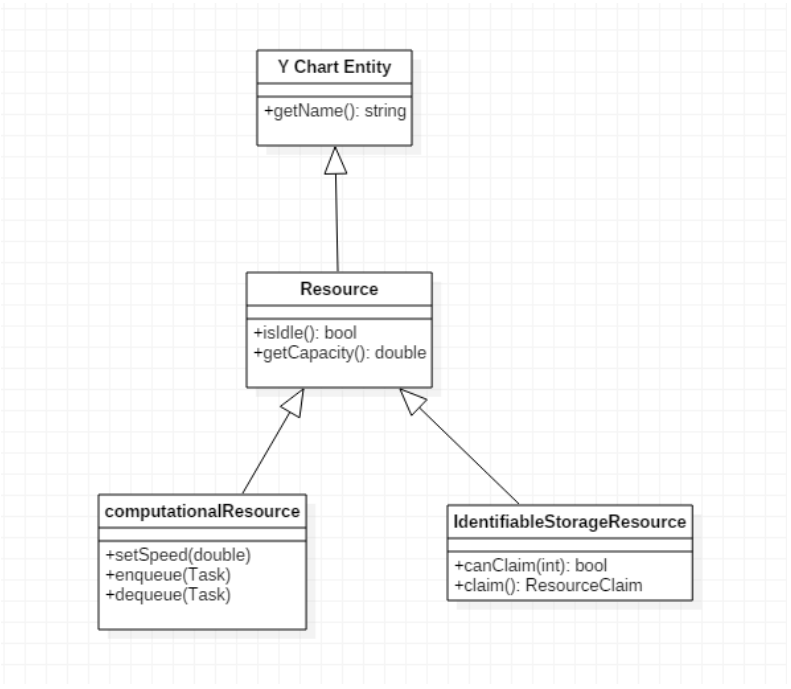


Figure 4.4: Resource Interface

4.2.1 Computational Resource

The **AbstractComputationalResource** class implements all the methods of computational resource interface. Every task will call the methods *enqueue*, *dequeue* methods of this class when it gets ready to run and when it has finished running. The communication between the task and resource is explained in the next section 4.3. The **SimpleTDMAResource** is responsible for setting the speed of the task when it is ready to run and resetting the speed when it has finished running with the help of *reschedule* method. An important thing to be noted here is that all computational resources in this model will be initialized with the object of *SimpleTDMAResource* by passing the name of the resource, whether usePriorities is enabled or not and the unitProcessing time of each resource respectively. The *usePriorities* would be enabled only if a resource is shared by multiple tasks and the *unitProcessing* is the amount of load a resource can serve for a time unit.

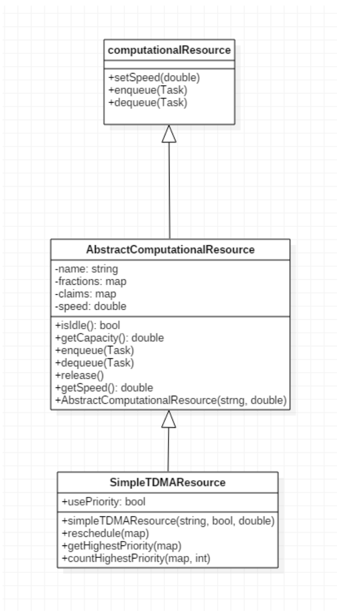


Figure 4.5: Computational Resource

4.2.2 Storage Resource

The **FirstFitStorage** implements the method of the IdentifiableStorageResource interface where a task would be ready to run only if it can claim the storage resource. In other words, the storage resource needs to have enough capacity to be claimed by the task; else the task would be blocked. After the above condition is checked, the *canClaim* acts as a flag to let the resource claimed by the task. The class diagram of storage class is shown below:

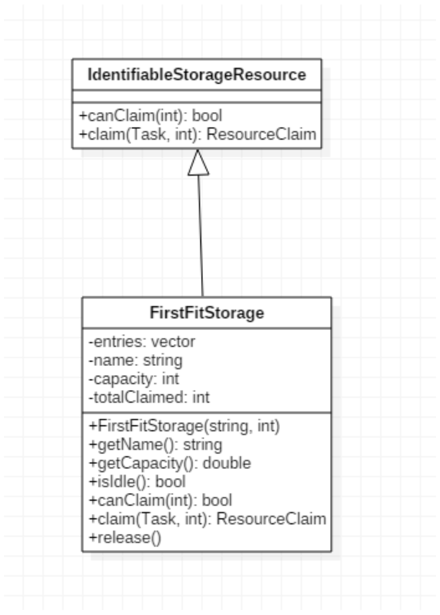


Figure 4.6: Storage Resource

The entire platform layer could be summed as following

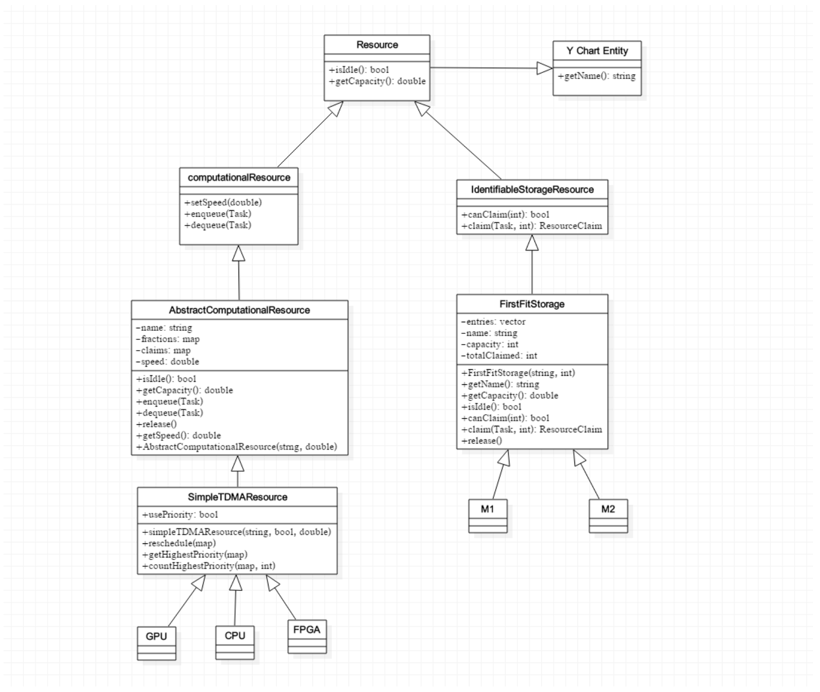


Figure 4.7: Platform Layer

4.3 Y Chart Layer: Mapping

This is the important stage in the Y chart methodology. It can be observed from the above class diagrams that the task gets mapped to the resource through four methods *enqueue(Task)*, *dequeue(Task)*, *claim(Task, double)*.This section throws light on the sequence in which these methods are called and their outcome.

4.3.1 enqueue

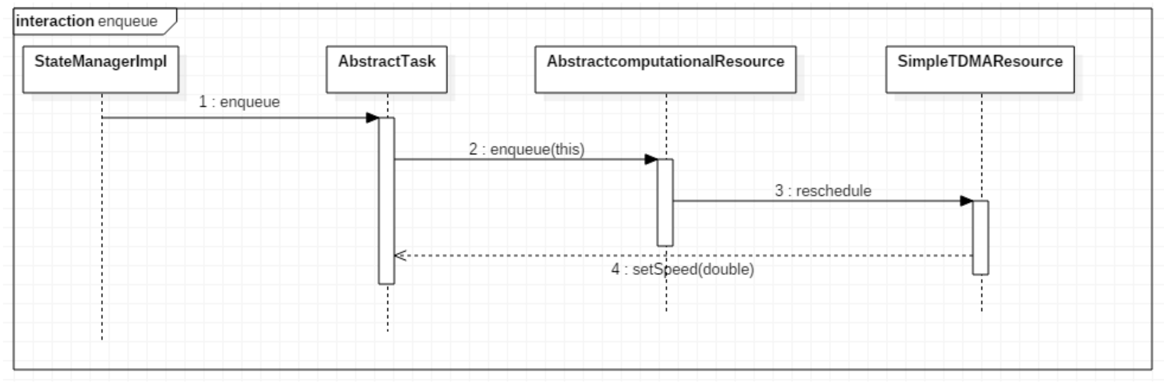


Figure 4.8: Enqueue Process

As explained in section [3], there are totally four states through which the task progress over time (NOT_READY, READY, RUNNING and BLOCKED). The *enqueue* method would be called in the NOT_READY state when the task gets enabled and becomes ready to run. Thus, the task enters the queue and the speed is set for the task.

4.3.2 dequeue

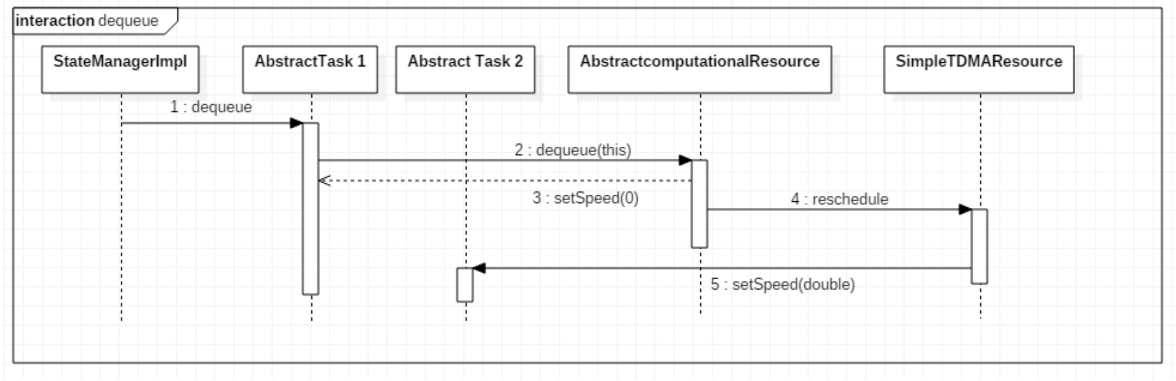


Figure 4.9: Dequeue Process

The dequeue method is called when the task has finished its execution.In case of dequeue, there is a need of reschedule method. For example, in a scenario where there are two tasks running on same computational resource platform. In that case, when one of the task fnishes the execution the speed of other task must be rest to full potential again. In this example, Abstract Task 1 and Abstract Task 2 share the same resource. Therefore, as Abstract Task 1 completes its execution, the dequeue method gets called setting its speed to zero and also setting back speed for Abstract Task 2 which can run with full speed again.

4.3.3 claim

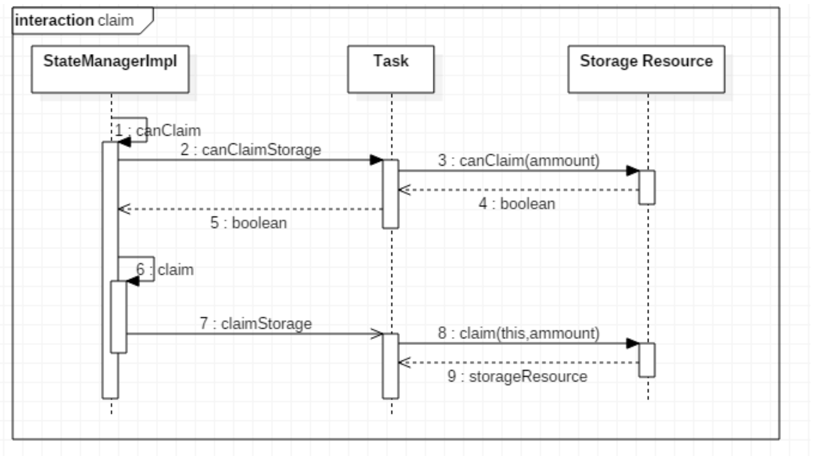


Figure 4.10: Claim Process

Every task when it moves from the state of NOT_READY to READY, it would be checked if it can claim a storage resource. Only when it can claim, it can proceed for the further course of action else it will be BLOCKED and it then needs to wait till it can claim a storage resource. In other words, it needs to wait till the capacity of storage resource if sufficient enough to accommodate it.

4.3.4 release

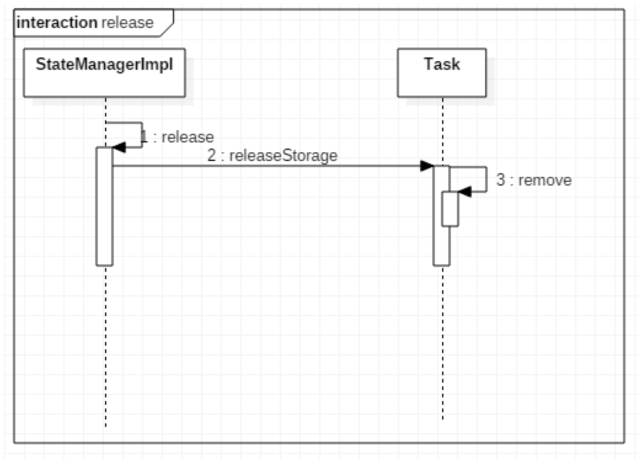


Figure 4.11: Release Process

The release method is called once the task finishes its execution. Normally when the releaseStorage method gets called, it removes the storage resource from the map which keeps track of the resource by using its corresponding name as the key. Then the data would be removed and added to the next task that must start. For example, in the example system Task B can start only after Task A has finished its execution. Thus, when the releaseStorage method of Task A gets called, it removes the data from itself and adds the same data(High, Low or Normal) to Task B and it carries on till the last task in execution (Task G in the example case) is reached.

Chapter 5

Execution Layer Implementation in SystemC

The next step in the implementation is the design of the execution layer in SystemC. For every task, a State Manager will be created for its progress from one state to another and this State Manager was designed as SystemC module. The example system as a whole had seven tasks, two storage resources and six computational resources. As a first step, a library was created. Then, a two task model was designed and implemented. Then, a five task model was designed and as the final step the example system was implemented using the library.

5.1 Two Task Model

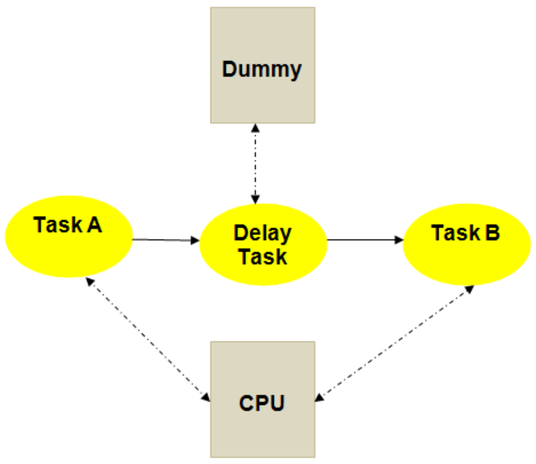


Figure 5.1: Two Task Model

The idea of designing a two task model was to have two different tasks sharing a same computational resource. Though it was chosen to be CPU in this example, it could also be a GPU or a FPGA. The idea behind is that Task A starts the execution at first and after sometime the Task B also starts executing. At this point, both tasks A and B will be running at same time sharing the same resource (CPU in this case). In such a case, the speed of the task would be set to 50% in this case (as only two tasks are sharing the same resource). If any one of the above two tasks finishes, the other task can start running at a full speed till it needs to share the resource again. The use of Delay task is to separate the start times of Task A and B. The Dummy resource was created for this delay task. The expected behavior could be visualized as follows:

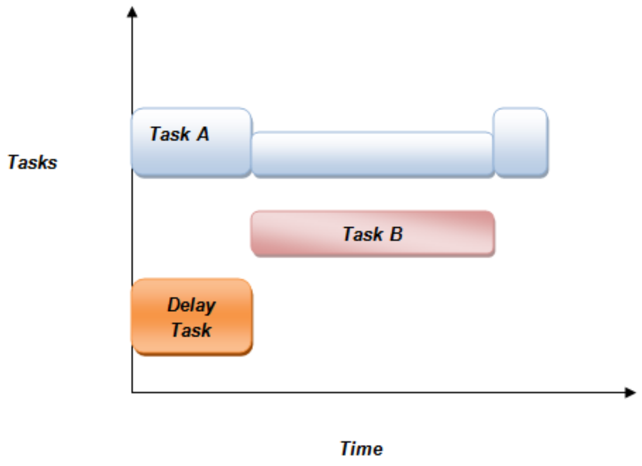


Figure 5.2: Scheduling of Two Task Model

The Task A and Delay task will start at time 0. After the Delay task finishes, the data will be sent to Task B to starts its execution. Two tasks (A and B) will run together with half speed and once any of the task finishes its execution, the other task will resume to its normal speed and this process could be repeated for number of times. The above system was implemented with Task A having a load of 200 (i.e.) it will take 200 time units for the task to finish as the processing time of CPU is 1 unit. Task B was chosen to have a load of 100 and the load of Delay task was chosen to be 20. The State Manager of a task was created as SystemC and since the usage of storage resources was excluded, the task only has three states which are namely : NOT_READY, READY and RUNNING.

1. Every task will start in a NOT_READY state. If the task possess a data, then it is said to be enabled.

2. If the task gets enabled, then it gets *enqueued* to its computational resource (CPUT,GPU or FPGA) and moves to RUNNING state.

3. In the RUNNING state, the task would be checked if it can claim a storage resource. This method would not be active in the above case as it does not use any storage resource. Hence it will move to the RUNNING state.

4. In the RUNNING state, the task would be executed and once finishing its execution, it would release all the storage resources, would be dequeued from the computational resource and moves to NOT_READY state till it gets another data to run.

5.2 Design Decisions

5.2.1 State Manager as SystemC Module

A State Manager is created for every task. The transition of the tasks from one state to another comes under the responsibility of the state manager. This was the main motivation of implementing the State Manger as the SystemC module. It was made to interact with the SystemC engine and used SystemC scheduler to process all the tasks which were created as SystemC threads respectively.

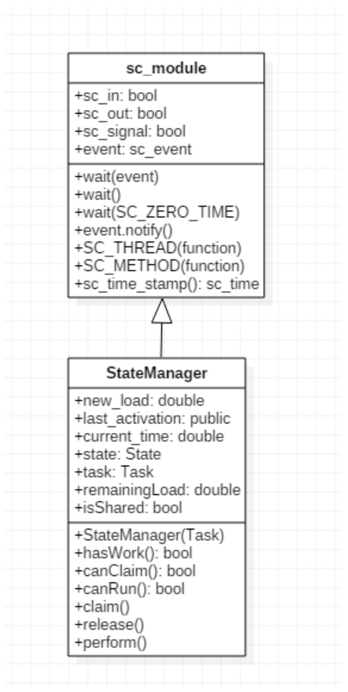


Figure 5.3: State Manager as SystemC Module

The class diagram 5.3 illustrates the attributes and the methods that were inherited. Only the methods/attributes used in the implementation were specified above as SystemC macro is huge and covering all the functionalities is out of scope. The method hasWork() checks whether a specific task has the object data to run in the NOT_READY state. The canRun() method checks whether the task has enough speed to run and canClaim() method checks whether a task can claim a storage resource or not. A task must satisfy both these methods in order to proceed to the RUNNING state. The perform() method performs all the transitions of a task from one state to another after running all the above methods.

5.2.2 Decision on Wait Statements

The condition for four states of a task (NOT_READY, READY, RUNNING and BLOCKED) was implemented using the case statement. Though BLOCKED state would not be encountered in this case of Two Task model (due to absence of storage resource), it would be encountered for later implementations. The important challenge was to use the suitable type of wait () statement within the SystemC thread. Every SystemC thread must be declared as a loop; else it will run for only a single time and terminate leading to undesired behavior. Hence two important factors play a role here: where the wait statements must be placed and which type of wait statements must be used.

Placing of Wait Systems

- Every task starts with NOT_READY state and it can move to the READY state only if it satisfies the condition of *hasWork()*. But, if it doesn't satisfy the condition, then the thread would never terminate causing a deadlock (it will be keep on looping in the NOT_READY state itself). This was avoided by placing a wait statement for the scenario where the *hasWork()* condition was not satisfied, thereby suspending the current task(thread) and starting the remaining tasks(threads) that are ready to run.
- As there is no storage resource used in this example, the *canClaim()* method would not be checked and hence if a task satisfies the NOT_READY state, it would go to RUNNING state from the READY state without any interruption.
- The RUNNING state is most interesting than all the other states as it defines the actual behavior for the simulation. Considering a scenario from the above example system where two tasks (Task A and Delay Task) starting at the same time. Considering Task A starting first, it would satisfy the *hasWork()* condition and gets enqueued to its corresponding computational resource (CPU in this case) where the speed of the task would be set and then moves to READY state. From READY state, it moves to RUNNING state where the required time for its completion would be calculated by dividing the load of the task to the speed of the task. At this point, the cooperative scheduling must happen, thereby giving the scheduler to the Delay task as it is also ready to run. Hence, a wait statement must be placed here thereby making the Delay task to move from the NOT_READY state to READY, then to RUNNING where the remaining time for the delay task would also be calculated. After waiting for a particular amount of time, the new load would be calculated to check whether the task had finished running or not (in such a case the new load will be zero). The task goes to NOT_READY once getting finished, thereby waiting for the data to start running again The State Manager cold be represented as follows:

Choosing the Wait Statement Type

Though the placing of the wait statement were straight forward to decide, the type of wait statement to be used plays a crucial role in proper behaviour of the system. As discussed in chapter 3, there are four ways of using a wait statement. Considering the case of NOT_READY state, the task/thread would encounter this wait statement when ever it doesnt have any data. There are only two cases in which this would be a case: when the task is not yet ready to start and the other scenario would be when the task has done with its execution. In both these scenarios, one wouldn't know how long the thread needs to wait. Hence, wait(Time_Period, Time_Unit) cannot be used making wait() or wait(event) the ideal choice. Once encountering these two wait statement, a thread can become active only if there is an event notification.

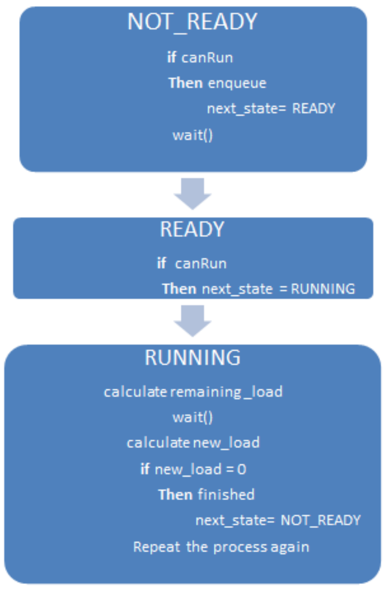


Figure 5.4: Algorithm of Two Task Model

```
case NOT_READY:
if(hasWork())
{
state = READY;
remainingLoad = task->getLoad();
task->enqueue();
run = true;
}
else
{
wait(event);
}
```

Listing 5.1: NOT READY state

```
if (canRun() && canClaim())
{
state = RUNNING;
claim();
run = true;
}
else if (canRun() && !canClaim())
{
state = BLOCKED;
task->dequeue();
run = true;
}
break;
```

Listing 5.2: READY state

As an event is needed to advance the simulation, the RUNNING state must also have a wait() or wait(event) system. Though, the amount of time a thread/task needs to wait could be obtained from the calcularion of remaining_time, using wait(Time_Unit,Time_Period) is not a wise idea as it will make the threads to behave in a strage manner. For example, considering a scenario where Task A and Delay Task are ready to run. The tasks will reach the RUNNING state and wait for their corresponding remaining time. The Task B which has no data to execute, will not satisy the *hasWork()* and will encounter the wait statement (waiting for an event). To make Task B resume its execution again, an event needs to triggered and a value need to be set for the activation of event. This in turn leads to a complex structure where one thread/task would wait for an event and the rest will be waiting for its own time making the simulation non synchronized thereby leading to a deadlock.

5.2.3 Design of Timer Module

An event is needed, as task in both NOT_READY and RUNNING states wait for the event to be notified, so that the time would be advanced and the remaining load could also be calculated. After all the tasks/threads enconter wait statement, it means they wait for an event. By Discrete Event Simulation priniciple, when a event is notified at least one of the tasks/thread must finish its execution respectively. In order to achieve this, a simple and straightforward strategy was followed:

- The remaining_time which indicates the amount of time needed for the completion of the task was calculated and stored in a C++ list/vector.
- After all tasks/threads gets suspended by the wait statement, the minimum value was obtained and an event notification occurs advancing the time for that minimum value.
- All the tasks/threads now wake up from the wait statement one after the other and the task/thread which had the minimum delay would finish.

This same prinicple can be used for the entire simulation till all the tasks/threads finish their respective execution and the simulation comes to an end. In this example, Delay task will insert its remaining_time value of 20 in the vector and Task A inserting its remaining_time value of 200. Since 20 is the least value here, the event notification will happen for it advancing the time by 20 time units, thereby the Delay task will now have zero load after the event notification and Task A will now have a remaining load of 180. If Task B also joins at this point, the same process would be repeated till the simulation finishes. One important factor to be noted here is that the vector/list must be cleared everytime a notification happens in order to prevent non deterministic behaviour and also for the fact that the size of the container will explode if the simulation is run for several hundred iterations. This principle is generic and can be applied to application with even larger number of tasks. Another interesting feature is both wait() and wait(Event) works similarly for this example case.

```
case RUNNING:

if (canRun() && !calculationPhase)
{
current_time = sc_time_stamp().to_default_time_units();
remainingLoad = task->getLoad();
remaining_time = remainingLoad / task->getSpeed();
new_load = remainingLoad;
calculationPhase = true;
speed = task->getSpeed();
}
if (calculationPhase)
{
remaining_time = new_load / speed;
delayy.push_back(remaining_time);
speed = (task->getSpeed());
last_Activation_here = sc_time_stamp().to_default_time_units();
wait();
current_time_here = sc_time_stamp().to_default_time_units();
new_load = new_load - (current_time_here - last_Activation_here)*speed;
run = true;
}
if (new_load == 0)
{
state = NOT_READY;
release();
cntt++;
calculationPhase = false;
run = true;
}
break;
```

Listing 5.3: RUNNING State

As the task enters the RUNNING state, the current time could be obtained from the SystemC datatype *sc_time_stamp().to_default_time_units()* which could be used to obtain the timestamp and store it in a variable. As mentioned before, the remaining_time representas the ammount of time equired for completion for the tasks current speed. The calculationPhase was created for a purpose. Considering a case of this example where Delay and Task A would run parallely. First activation would be made for 20 time units. At this time, the load of Delay Task woud be finished (as the load would become zero). But, Task As remainng time needs to be calculated again as the Task B would arrive and the speed of Task A would change. To avoid such a case, a *claculationPhase* was used to calculate the remaining time of every task during the course of the simulation till the task finishes disabling this flag.Thus, after calculating the remaining_time, the value gets added to the vector delayy and then the wait(event)/wait() statement would be encountered.

```
double Time_Period = 0;
if(!delayy.empty())
{delayy.sort();
Time_period = delayy.front();}
event.notify(Time_Period,SC_NS);
```

Listing 5.4: Timer Module

The Timer module checks if the vector which keeps in track of *remaining_time* of all the tasks is empty. If it is empty, it notifies for zero time (no advance in time) else it sorts the vector and chooses the least value. The event then notifies for this value and advances the time. Then State Manager resumes and calculates its remaining load and this process gets repeated till all the tasks finish its execution. The interaction between the State Manager and Timer could be summarized as following
The State Manager and the Timer can communicate with each other in two ways. One is by using ports and signals and other would be wihout using a port or signal rather by having

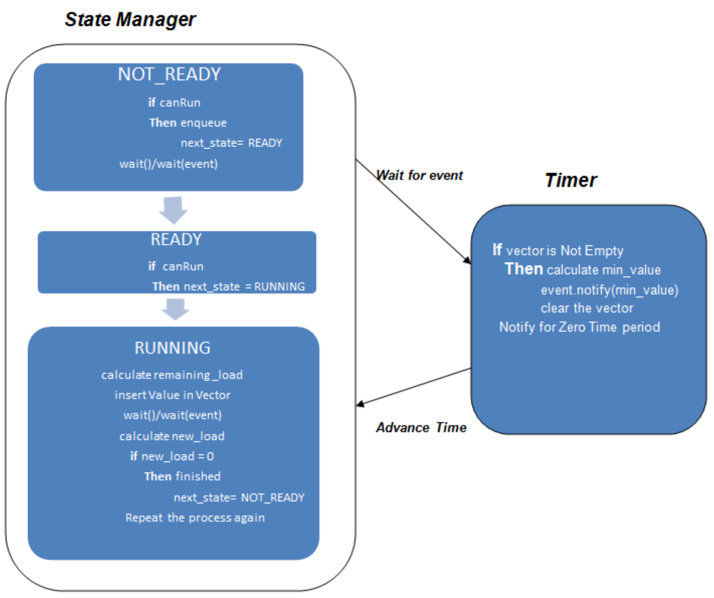


Figure 5.5: Interaction Diagram

the Timer as another SystemC thread or SystemC method. The algorithm was implemented in both the above mentioned ways and is explained briefly in the following sections.

5.2.4 Implementation With Ports

For any system, the State Manager should communicate with the Timer Module to advance the time. The State Manager and the Timer was considered as separate SystemC modules. Since the State Manager is the module which waits for the event from the Timer to be notified, every State Manager would have an output port sending a signal for an event notification and then the Timer module needs to have an input port to receive the signal.

Array of Ports

Though a single port is sufficient for every State Manager to send the signal, the Timer needs to receive the signal from all the State Managers. In other words, if there are seven tasks then seven State Managers will be created for these tasks and they would signal the Timer Module. So the Timer module needs to receive signals from all these State Managers. Array of ports feature in SystemC suits the requirement for a single SystemC module to have multiple ports to send or recieve different signals respectively. By specifiying the number of tasks in the system, the array of required number of ports would be created for the Timer module. The sytax for the array of port is as follow:

sc_in< T >Port_Name[Size]

The array of input ports is given as an example as this technique is being used in the implementation. Using the same methodology, array of output ports could also be created. These array of ports from the Timer can be connected to the ports of the State Manager by using the array of SystemC signals defined as follows

sc_signal< T >Port_Name[Size]

The important thing here is that the State Manager module must be made sensitive to the event (by specifying sensitive statement and the event name) to sense the advance in time after the event notification.

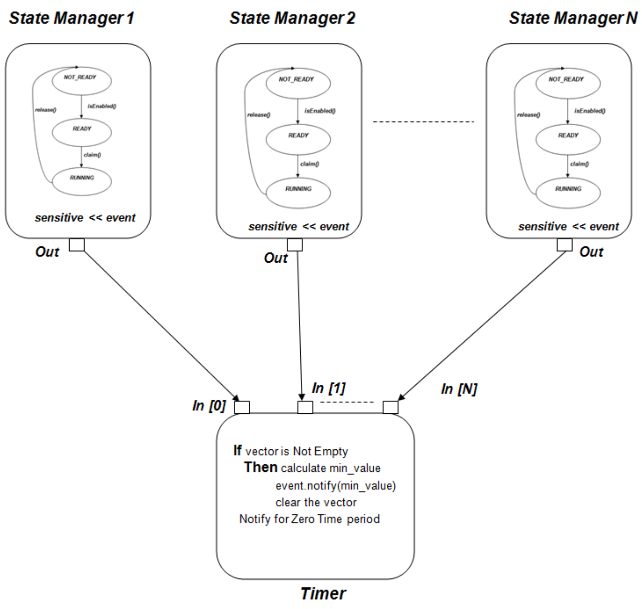


Figure 5.6: Implementation with Ports

The figure 5.6 represents the implementation of the system. The functions inside the State Manager module is a SystemC thread as the tasks needs to run parallely by waiting for each other. But, for the case of Timer module, the function to notify the event was implemented as process (SC_METHOD) instead of a thread. The motivation for this decision came from the fact that, during the course of simulation the State Manager of each active task would run and then would wait for the Timer to advance the time. There is no need for the Timer to wait now as it just checks the input ports for the signals from the task and then notifies an event.

```
sc_in<bool> input[portSize];

void timer()
{
double Time_Period = 0;
if(!delayy.empty())
{
delayy.sort();
Time_period = delayy.front();
}

}

for(int i=0; i< portSize;i++)
{
if(input[i].read())
{
```

```
event.notify(Time_Period,SC_NS);
}
```

Listing 5.5: Timer Module With Array of Ports

5.2.5 Implementation without using Ports and Signals

The implementation without ports or signals was not as straight forward as the implementation with the ports and signals. The reason is because the process or thread doesnt have a specific order of execution in SystemC and if these are not synchronized properly, then the simulation would lead to a deadlock. The first approach which would seem very obvious in the implementation is to have two methods in the same module. For example, the State Manager module would have both the *perform()* method and *timer()* method. In this case, a choice must be made for the *timer()* method to be a SC_METHOD or a SC_THREAD. But, both of these choices will not work. The reasons is when a two or more functions are defined under the same module, the order in which they would be running is unknown which would in turn lead to non-deterministic behavior and failure of the simulation as both the processes would be sensitive to the same event.

```
void perform()
{
switch(state)
{
case NOT_READY:
if(hasWork())
{
/* Do some Processing*/
}
else
{
wait();
}
break;


case READY:
if(canRun())
{
/* Do some Processing*/
}
break;

case RUNNING:
if(canRun())
{
/* Calculate the Remaining Time*/
wait();
/* Check for the New Load*/
}
break; }}

void Timer()
{
double Time_Period =0;
if(!delayy.empty())
{
delayy.sort();
Time_period = delayy.front();
}
}
```

Listing 5.6: Module with Multiple Processes

In the above example, if *Timer()* function starts first, the *Time_Period* would be zero as the vector would be empty. Hence it will notify the event ev1 for zero time and as *Timer()* method is also in the same SystemC module which is sensitive to ev1, it will execute again. This will repeat forever without allowing the *perform()* to run thereby leading to deadlock. The important factor to be noted here is that when *Timer()* is defined inside the StateManager class, then this method would be executed for each task there by notifying several times with different vales and corrupting the time stamp. Using the *Timer()* as SystemC threads would also not work in this case. For example, if it is made as a SystemC thread, then all threads needs to have only wait(event) as using wait() statement only waits for the event that notifies first corrupting the time stamp again. Another disadvantage is that the design choice would now get limited as the wait(event) can only be used. This problem was solved by having Timer() in separate SystemC module. As no ports or signals would be put to use, establishing communication was difficult. But, this was solved by the SC_ZERO_TIME feature. As discussed, this makes a process(SC_METHOD) execute with a delay of zero time units or a thread to wait for zero time units without affecting the behaviour of the system.

If the State Manager module starts first then all the tasks/threads would be executed till wait statement is reached and the control would be passed to Timer module where the event notified for it is zero time units, hence it performs the necessary activation and when the threads in the State Manager wake up to the advanced time and the simulation would proceed till all the tasks/threads finish the execution. It is to be noted that the same Timer() function could be implemented as SC_METHOD as the function is now sensitive only to the event1 and it won't run forever and also SC_METHOD has an advantage over SC_THREAD as SystemC threads performs context switching and would be slower when compared to the SC_METHOD.

5.2.6 Avoiding Race Condition

Task A and Task B share the same resource CPU. Therefore, the speed of the tasks must be changed dynamically during the course of the simulation. Since, the order of execution of threads doesnt follow a specific order; the system has a high risk of race condition. In the design, the speed of the task would be calculated before encountering a wait statement and also after waking up from the wait statement to check for the speed change (in case of arrival of new task sharing the same resource). Then the new load of the task would be calculated. An example scenario for the race condition to not happen is as follows: Task A and Delay Task starts simultaneously and after the first event notification Delay Task would get over. Now, Task A would have calculated its new remaining load. The new load would be calculated using the following formula.

$$\boxed{new\_load = remainingLoad - (current\_time - last\_activation) * speed}$$

As the Delay task finishes, it would send the data to Task B enabling it to run. Thus task B would come till the wait statement in the RUNNING state and gives the signal to Timer module to find the least delay and perform event notification. Only if the Task A wakes up first from the wait statement before Task B, it could calculate the new load with the speed of 0.5 and recalculate the time it needs to complete thereby leading to successful finish of the simulation. But, if Task B wakes up from the wait statement first, then it would calculate its new load and since it would become zero (as task B would have the least delay) it would finish thereby releasing the resource and updating the CPU speed to 1 instead of 0.5. As B finishes, Task A would start and the new load calculation of it would become wrong now as the peed is no longer 0.5 thereby causing the simulation to fail. This makes this algorithm non generic as it depends on particular order of execution which is not possible with SystemC engine.

This problem was solved by following a straightforward methodology. Every task has a flag named *speedChange*. When a Task resumes its execution from the wait statement in the RUNNING state, its speed will be checked. If the speed is lesser than one then it means the resource is being shared and hence the speedChange flag will be set. Even if the Task finishes its execution, before releasing the resource it checks for the *speedChange* flag and if it is set, it waits for zero time units through the statement wait(SC_ZERO_TIME). This in turn will ensure that the other running tasks which share the same resource have its new load calculated correctly with the proper speed thereby eliminating the race condition and making the design generic.

```
CASE RUNNING:

..............
wait();
new_load = remainingLoad - ( current_time- last_activation)* speed;
if(speed < 0)
{
speedChange = true;
}
..............

if(new_load == 0)
{
Finished = true;
if(speedChange)
{
wait(SC_ZERO_TIME);
speedChange = false;
}

release();
```

Listing 5.7: Avoiding Race Condition

5.2.7 Five Task System

This system consists of five tasks and it also uses a storage resource. It has the same working characteristics of the example system. The system is schematically represented in figure 5.7. Both the computational resource and storage resource could be shared by one or more tasks. By designing the following system, the sharing of storage resources and dealing with BLOCKED state was also performed making the library generic which could be extended for implementing the main example system (data path case of a printer) and any data intensive application with the similar behaviour.

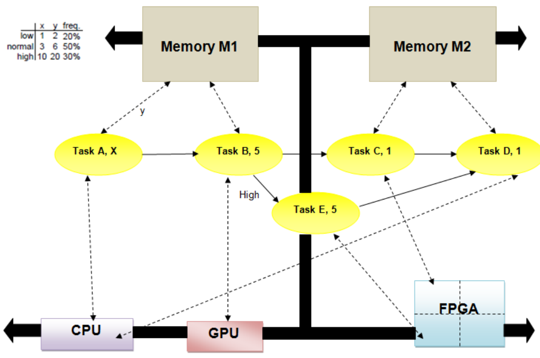


Figure 5.7: Five Task System

The dynamics of a system like this is already explained in section 3.4.1. The most important thing to be considered here is that if two Tasks share a same storage resource and at one point if the resource cannot accommodate a task, then it sends the task to BLOCKED state and the task waits till the resource gets available again. It is clear from figure 5.7 that Task A and B share the same storage M1. The storage resources gets released when ever the tasks finishes its execution. For example, when Task A needs to start its execution and cannot claim the storage resource because it is already used by Task B and the claim amount by Task A exceeds the amount which is free in the storage device, then it goes to the BLOCKED state. A special care must be taken again in the BLOCKED state as the SystemC thread would again infinitely remain in the BLOCKED state causing a deadlock. This can be prevented by having a wait statement in the BLOCKED state. The new state diagram is as follows

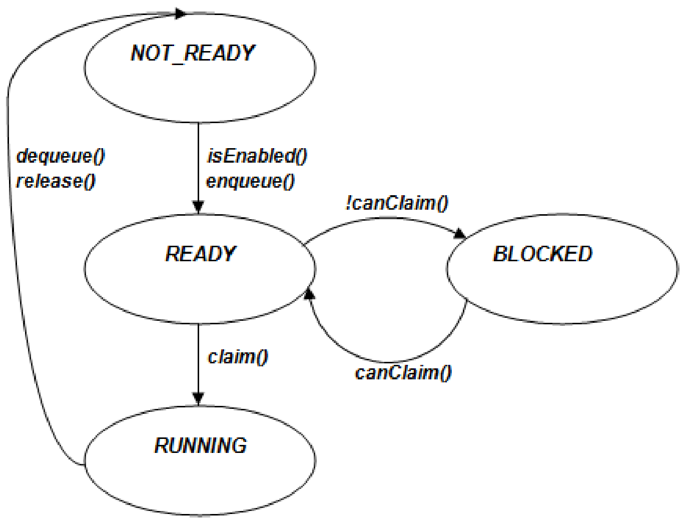


Figure 5.8: Automaton with BlLOCKED state

It is clear from the figure 5.8, the *canClaim()* plays a major role here. It will be used in READY method to check whether a task can claim a storage resource, if it gets satisfied the task moves to the RUNNING state, else it moves to BLOCKED and waits till it can claim the resource again. The algorithm followed is explained in the following diagram

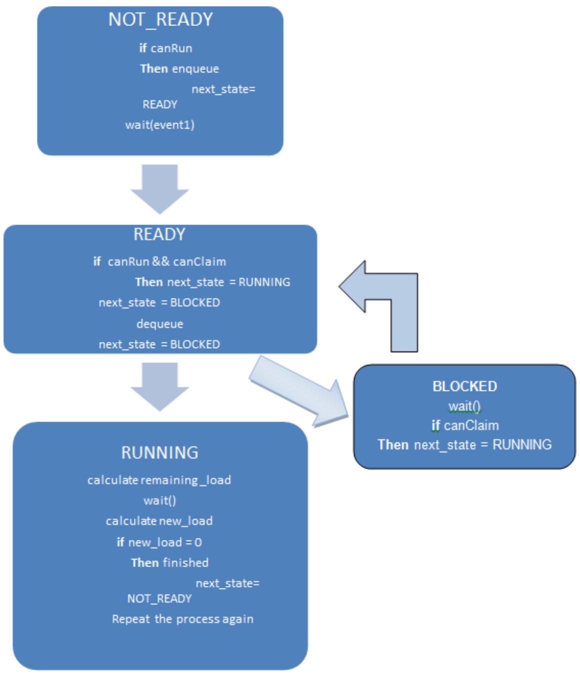


Figure 5.9: Algorithm with BLOCKED state

It can be noticed that the NOT_READY state has a wait(event) statement and RUNNING, BLOCKED state has wait() statements. In the case of Two Task example, both wait() and wait(event) were having the same effect as the order of execution was deterministic and only maximum of two processes was running at any time. But, it won't be the case with respect to the five task model or any other model which uses the storage resouce. In case of Five Task example, if the NOT_READY case has the wait() statement instead of wait(event) statement, then after a notification of event from the Timer module, the tasks/thread which doesnt have any data to run would resume from the wait statement in NOT_READY state and go back to wait again. The task which is active resumes itself from the wait statement, finishes its execution and would release the data for the next task. Since the next task to which the data is being passed already ran once, the task would not become active thereby causing simulation failure. For example, in a system with a input of 10 objects, by design the Task A0 is active and all the others are not active at the beginning of the simulation. Now, as A0 moves to RUNNING state and encounters the wait statement, all the other tasks/threads would be checked for its input. Since none of the tasks/thread would have input, they would go back to wait state. Now the Timer module would run and would advance the time for the corresponding time unit. If we use wait() in NOT_READY state, then all the tasks/threads which were not active before would resume and check for its input. As there is no input yet, they would return back to wait. After this, A0 would resume and it would eventually finish thereby giving an input to B0 and also starting A1. Now, A1 and B0 need to execute parallel. As B0 already encountered a wait statement before, it would not run parallel allowing only A1 to run. This in turn would cause a domino effect across the starting time of all the tasks resulting in simulation failure. It was also observed that if a task/thread encounters a wait()statement and if another task encounters wait(event) statement, after the event gets notified, the task/thread which encountered wait() statement resumes from the wait first. But, if two tasks/threads encounter wait() statement or wait(event), then the order in which they resume their corresponding operations are unknown. Therefore keeping all these observations under consideration, the above algorithm was designed thereby making it generic and not depending on a specific order of execution. This approach was tested for the data path example system and the output was verified using the TRACE software respectively.

5.2.8 Randomizing Storage Memory Access

During the course of simulation, it was explained that a task would go to BLOCKED state if it doesnt have the capability to claim a storage resouce. With this type of scenario, let us assume Task A starting with the simulation first, then it would claim the storage resource and proceed to RUNNING state. Now, when Task B start its execution, it would not have the capability to claim as most of the storage space is occupied by Task A. Hence, this would make the Task B to move to BLOCKED state. After an event notification, Task A would complete its execution, release the resource and again starts the execution of its next input object while Task B would be in the BLOCKED state till all the input objects of Task A finsihes its execution. Due to this factor, in all the cases a specific Task would only be BLOCKED which is very evident from the trace below.

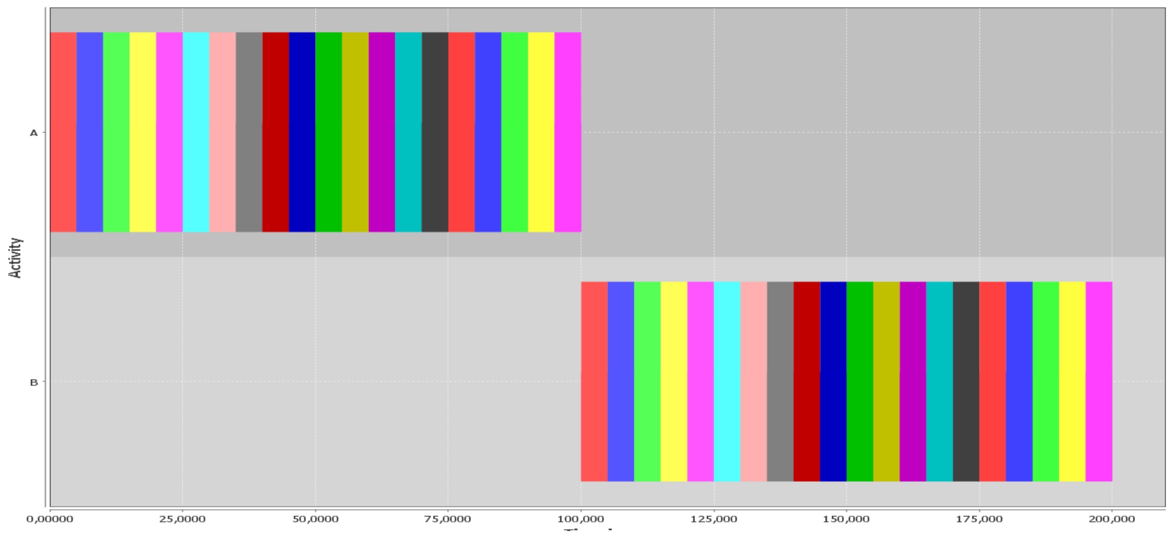


Figure 5.10: Output Trace Without Randomness

Instead of blocking Task B everytime, a procedure was analyzed where both the Task A and Task B randomly access the storage resource M1. In other words, a specific task would not be blocked always. This randomization was adopted through following steps

1. For every task, the storage resource gets claimed in the READY state.
2. A C++ random generator was used in order to generate a very small random number (between 0.001 and 0.002) and a seed was generated.
3. A wait statement was placed before checking the *canClaim()* condition in the READY state.

4. As the simulation starts, both the tasks start running parallely.
5. When these tasks reaches the READY state, a random seed would be generated for each of them and they wait correspondingly with the obtained random seed.
6. The Task which has the least wait time would continue the simulation and the other task would be blocked.
7. As the running task finishes its execution, it would start with the next input object and would wait in the READY state. At this moment, the task in the BLOCKED start will be eligible to claim and it would also move to READY state where the random seed would be calculated and the tasks would wait for their respective random seed repeating the process again till the end of the simulation.

Thus, the acquisition of the storage resource was made random between the two tasks and the following trace was obtained.

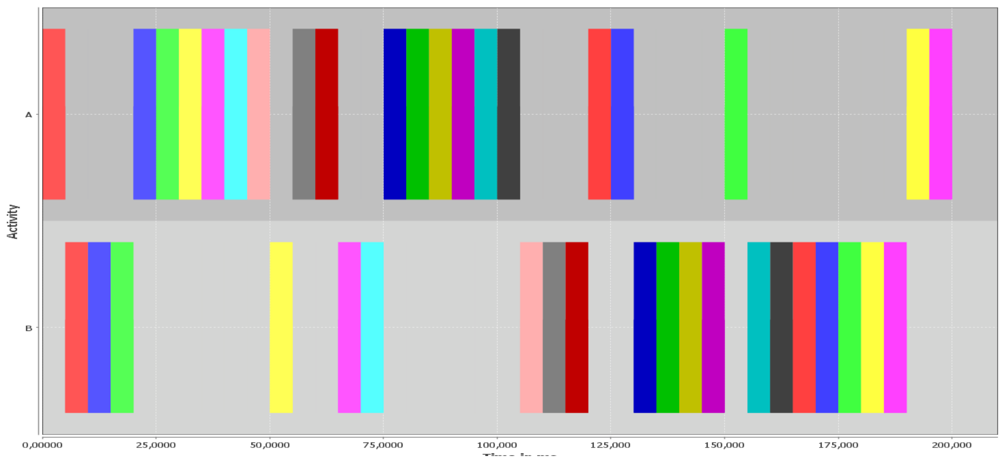


Figure 5.11: Output Trace With Randomness

The wait statement used here was not wait(random_seed, Time_unit). The reason is the same factor as mentioned in previous chapter. When a wait(random_seed, Time_unit) alone is used in a simulation for a system like above, it would cause the simulation to fail. In this case if Task A proceeds to RUNNING state with the minimum random seed of the two, then it will encounter the wait sattement. At this moment, Task B needs to move to the BLOCKED state before an event notification occurs. But this would not happen with wait(random_seed, Time_unit) as it would not allow Task B to proceed to BLOCKED state instaed making Task A to run infinitely as the wait statement in RUNNING state is the wait to an event notification and since because wait(random_seed, Time_unit) is not waiting for an event, this would make Task B to be never active again. This was overcome by using wait(random_seed, Time_unit, event) statement in the READY state. This would allow the Task B to go to BLOCKED state before the event notification happens thereby achieveing the expected behaviour. This was also made generic enough such that it was adopted in the example system (with 7 tasks) and the simualtion results were verified using the TRACE software.

5.3 Alternative Implementation in C++

This implemenattion was similar to that of Java where the simulation layer was written explicitly and the performance was measured in order to benchmark it with the results of SystemC and Java. The class diagram of simulation layer is almost the same as Java implementation in[section 2].

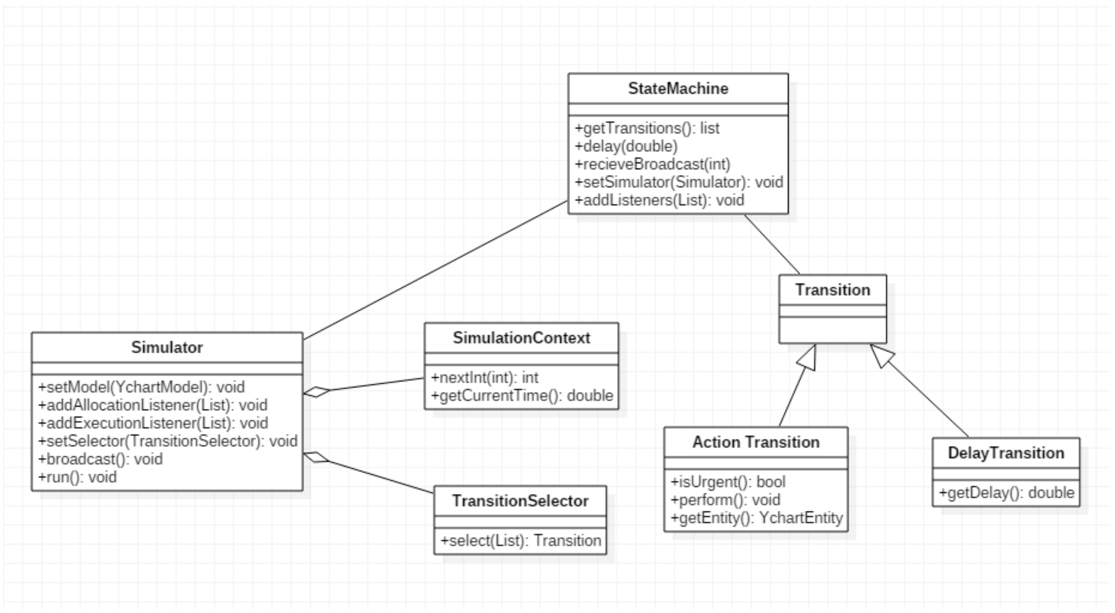

StateMachine
+getTransitions(): list
+delay(double)
+recieveBroadcast(int)
+setSimulator(Simulator): void
+addListeners(List): void
Transition
Simulator
+setModel(YchartModel): void
+addAllocationListener(List): void
+addExecutionListener(List): void
+setSelector(TransitionSelector): void
+broadcast(): void
+run(): void
SimulationContext
+nextInt(int): int
+getCurrentTime(): double
TransitionSelector
+select(List): Transition
Action Transition
+isUrgent(): bool
+perform(): void
+getEntity(): YchartEntity
DelayTransition
+getDelay(): double

Figure 5.12: Smulation Layer

5.3.1 Simulation Layer Progression

The message exchanged between the different classes is represented in the figure 5.13. The progress of simulation is as follows:

1. The computational resources (CPU,GPU,FPGA) and storage resources (M1,M2) were created.

2. The tasks were created and constructed with the resource it will be mapped to and the priority of each task.

3. The tasks and Resources are added to Y chart Model.

4. The simulation then starts with run() method.

5. As a first step, the State Managers were created by obtaining the Tasks and the resources to which it was mapped to from the Y chart Model.

6. Then the actions were obtained using getActions() method. The *actionList* would be retured with the list that contains the transitions that are pending to happen like NOT_READY_TO_READY, READY_TO_RUNNING etc.

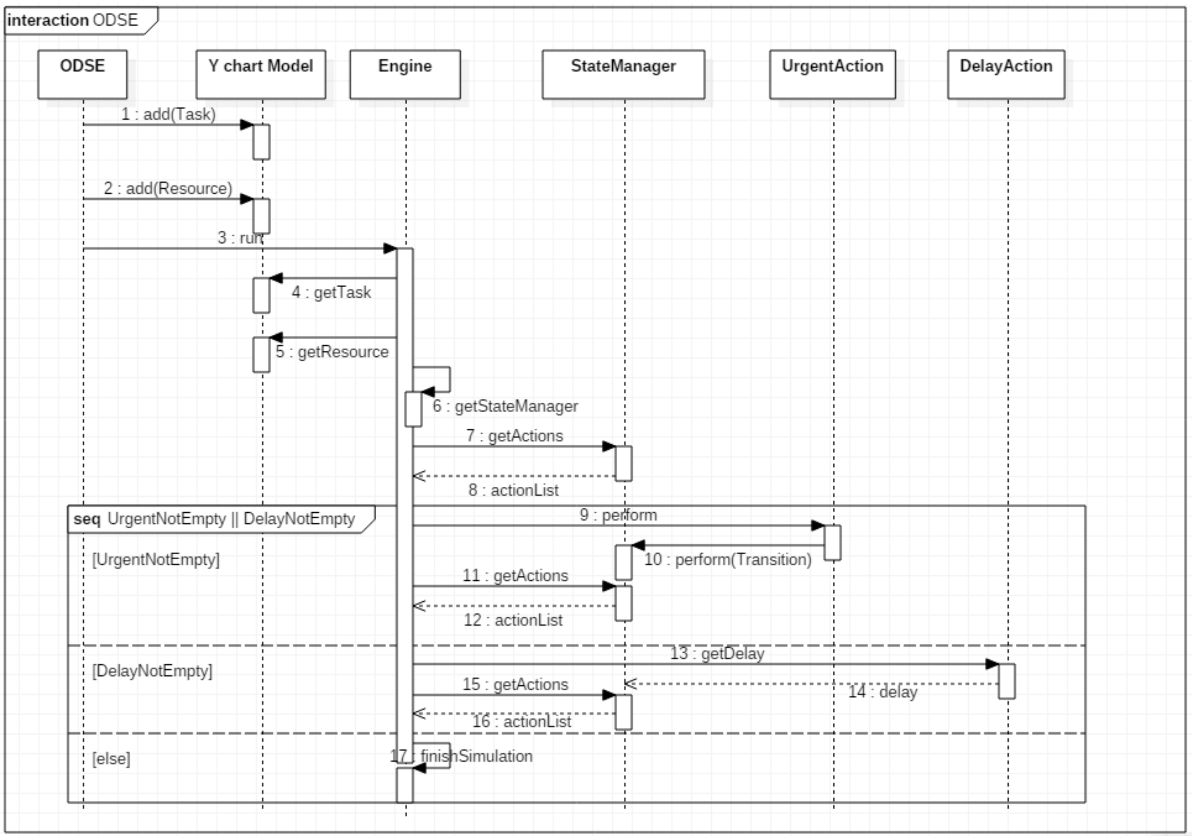


Figure 5.13: Simulation Layer Progress

7. After checking whether the obtained actions are Urgent actions (transition from one state to another state) or delay actions(the advance in time by a specific task), the corresponding action would be performed.

8. The urgent actions include the progress of task from one state to another like the transitions from NOT_READY to READY state, READY to RUNNING state etc. This would be performed till there are no urgent actions.

9. After this, the delay action would be performed by advancing the time depending on the task that was finished and updating the time in state managers of all the other tasks.

10. This process would be performed till there are no more urgent or delay actions to be performed.

11. As the action list becomes empty, the simulation comes to an end through *finishSimulation()* method.

Chapter 6

Evaluation and Conclusions

6.1 Evaluation

6.1.1 Simulation Speed

By definition, *the performance modeling is the process of giving an input or a stimulus to a system and measuring the time the system takes to produce the output under various load conditions*. Therefore, the simulation speed is the primary parameter in terms of measurement. The Java library with the example system (7 Task Model) was available already. Using the library, a Two Task model was designed and a 5 task model was also designed. Similarly, a Two Task model, Five Task Model and Seven Task Model were also designed from the library created using SystemC (using ports and signals), SystemC (without using Ports and Signals) and pure C++ Implementation. The input objects were fed to these systems to measure the performance under increasing load. The number of input objects that were fed to the system was from 10 to 100,000 and the results were recorded and analyzed.

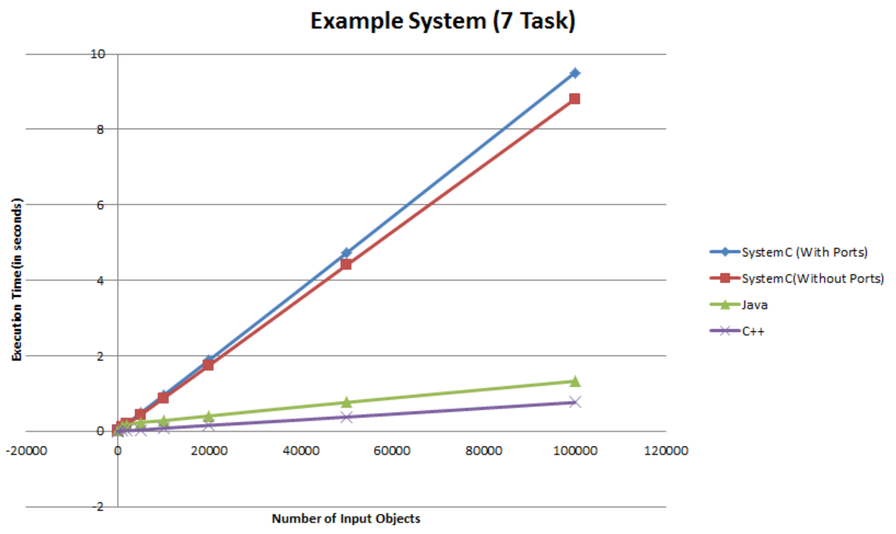
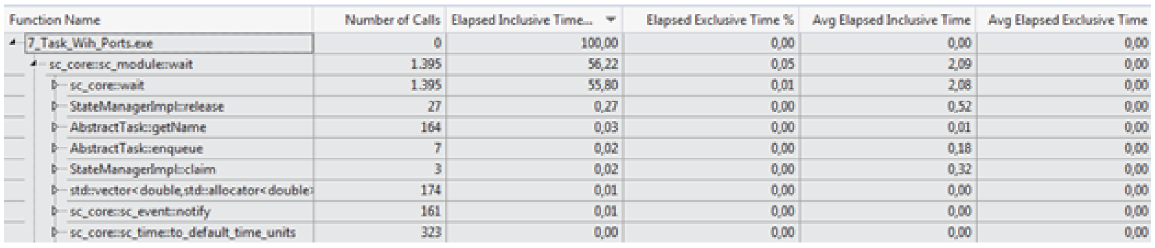


Figure 6.1: Overall Simulation Speed

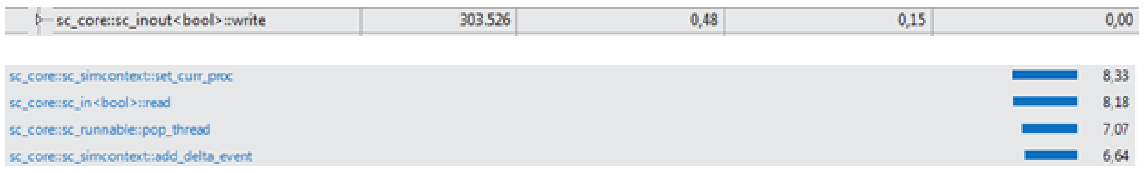
The SystemC with ports is the slowest of all the implementation (nine times slower than the Java implementation and almost eighteen times slower than C++) because of the context switching which is explained in this section. The wait statements play the major role in slow down. Whenever a thread gets suspended by a wait statement, all its variables need to be stored and when it resumes the activity, the parameters needs to be updated and the advance in time must also be taken into account. In the example system, there are totally four wait statements being used. In addition to that, these wait statements are called more often when tasks/threads run in parallel. This was the bottleneck with respect to SystemC. This was also confirmed by running the Visual Studio Debugger under the Instrumentation mode. The use of Instrumentation methodology of profiling is to find the detailed timing information about each function call



Function Name	Number of Calls	Elapsed Inclusive Time...	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time
7_Task_Wih_Ports.exe	0	100,00	0,00	0,00	0,00
sc_core::sc_module::wait	1.395	56,22	0,05	2,09	0,00
sc_core::wait	1.395	55,80	0,01	2,08	0,00
StateManagerImpl::release	27	0,27	0,00	0,52	0,00
AbstractTask::getName	164	0,03	0,00	0,01	0,00
AbstractTask::enqueue	7	0,02	0,00	0,18	0,00
StateManagerImpl::claim	3	0,02	0,00	0,32	0,00
std::vector<double,std::allocator<double>	174	0,01	0,00	0,00	0,00
sc_core::sc_event::notify	161	0,01	0,00	0,00	0,00
sc_core::sc_time::to_default_time_units	323	0,00	0,00	0,00	0,00

Figure 6.2: Function Call

The above data was collected for a Seven Task model using ports with 100 objects fed as input. As it is clear from the profiler reults that almost 60% of the time gets spent on the context switching through wait statement and the rest of the time was for the function calls having C++ maps and other containers which wouldnt play a major role in simulation performance when the optimization flags are enabled and the code is run in the *Release* mode. As the number of input objects increases, the number of calls to wait statement would also increase thereby increasing the simulation time lineraly. The ports and signals also contribute to the slowdown.



sc_core::sc_inout<bool>::write	303.526	0,48	0,15	0,00

sc_core::sc_simcontext::set_curr_proc	8,33
sc_core::sc_in<bool>::read	8,18
sc_core::sc_runnable::pop_thread	7,07
sc_core::sc_simcontext::add_delta_event	6,64

Figure 6.3: Function Call : Read and Write

For example, in the case of Two Task With Ports implementation, the model was run for 100 objects and the data were collected. It can be observed that the write doesn't take much time due to the fact that each State Manager has only one port and writing the value to it doesnt take time. But, it is not the case for read as the Timer module needs to read from the array of ports (3 in this case) and it needs to notify the event. This time would also increase with increase in number of ports being another reason for slowdown. This explains the reason behind SystemC implementation without ports being faster than the implementation with ports and signals. But it must also be noted that the SystemC implementation without using ports and signals is only slightly faster than the implementation with ports due to the fact that there must be an event synchronization mechanism again as explained in chapter 5 by delaying the Timer by using SC_ZER_TIME or by using a wait statement again if the *timer()* method was run as SystemC threads.

It was expected that C++ would outperform Java and hence the results were according to the hypothesis. After implementing the C++ version exactly as Java, it was having equal speed of Java and not faster as expected. But some optimiation techniques were followed by using constant refernces and passing pointers to function which gave a considerable speed up and therefore the C++ implementation developed is now twice faster than Java. The decision of designing Y chart layer in C++ and State Manager in SystemC played a crucial role here as the performance bottleneck for a data intensive application like a data path of a printer case was able to be identified effectively with the context switching being the cause of the slowdown.

6.1.2 Scalability

In this section, the speed of the implementation for increase in size of the system is analyzed. As discussed in previous section, three models were created. One is a model with two tasks and a signle resource, then a five task model which almost mimiced the example system and as a final step example system was also created. All these three models were implemented both using ports and without ports. The models with ports were analyzed for a input ranging from 100 to 1000 in order to study the time spent on communication between the State Manager module and the Timer Module.

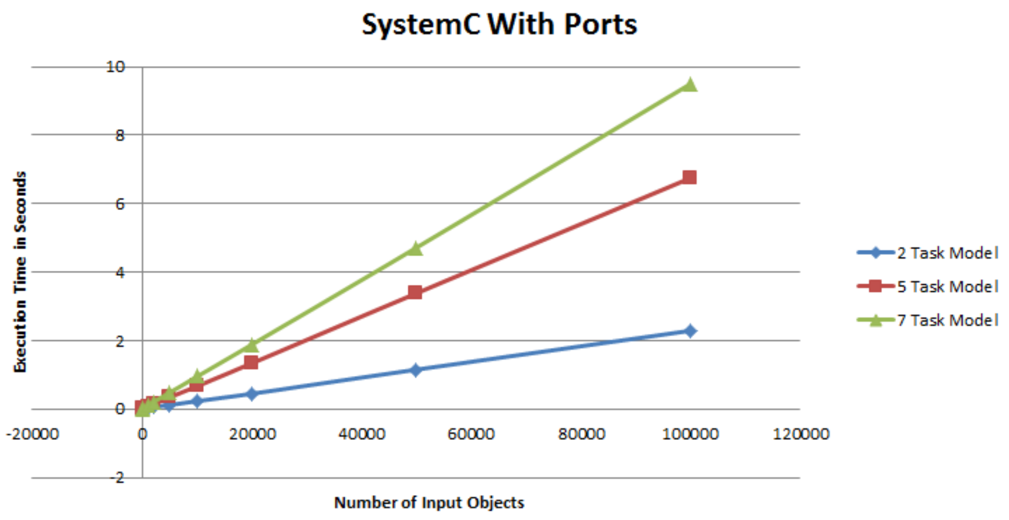


Figure 6.4: Scalability of the System

It is clear from the graph that the system gets degraded in performance as it gets scaled up with multiple tasks. The obvious reason is the increase in effect of the context switching by the wait ststements and also the time taken to read and write the SystemC ports. One interesting point to be noted here is that what ever the size of system is made to run, the speed of the simulation decreases only in linear fashion.

6.1.3 Ease of Modeling

Unlike languages like C++ and Java, the biggest problem SystemC faces is the documentation. Though the Language Reference Manaual (LRM) of SystemC is extensive and covers almost every aspect of the language, the examples which were given were abstract making the learning process complicated. Also the online resources (or) forum to discuss certain roadblocks is very limited for SystemC. In fact there is only one active forum in order to discuss about the language. Because of these factors, it consumes a considerable amount of time to understand the dynamics of the language by trial and error basis.

Debugging in SystemC is also difficult. It is due to the fact that most of the error messages or the warning can never be identified during the compilation process. Considering a scenario where two modules communicate with each other using ports. If either of the module gets designed without an input port, then a error at compile time is expected. But in case of SystemC, the compilation would run successfully and would fail during simulation with a error message "Complete Port Binding failed". Then it can be made to work by creating the corresponding port. This is just a sample case. When building bigger models, it would be little cumbersome to find the place where the error has occured and to rectify it. Modeling with threads and events made the process tedious as all threads needs to be synchronized to a specific event. For the example system, it had only four states. For system with more states, the modeling would also be more difficult due to more wait statements in the system. Since the entire simulation is run by the SystemC scheduler, special coding strategies must be made in order to achieve certain requirements. Having a flag "speedChange" to prevent the race condition of task with respect to the release of the computational resource is an example. Another example is the strategy of using the C++ random generator to make the tasks/threads to access the storage resource. The Learning Curve in modeling in SystemC could be represented as follows:

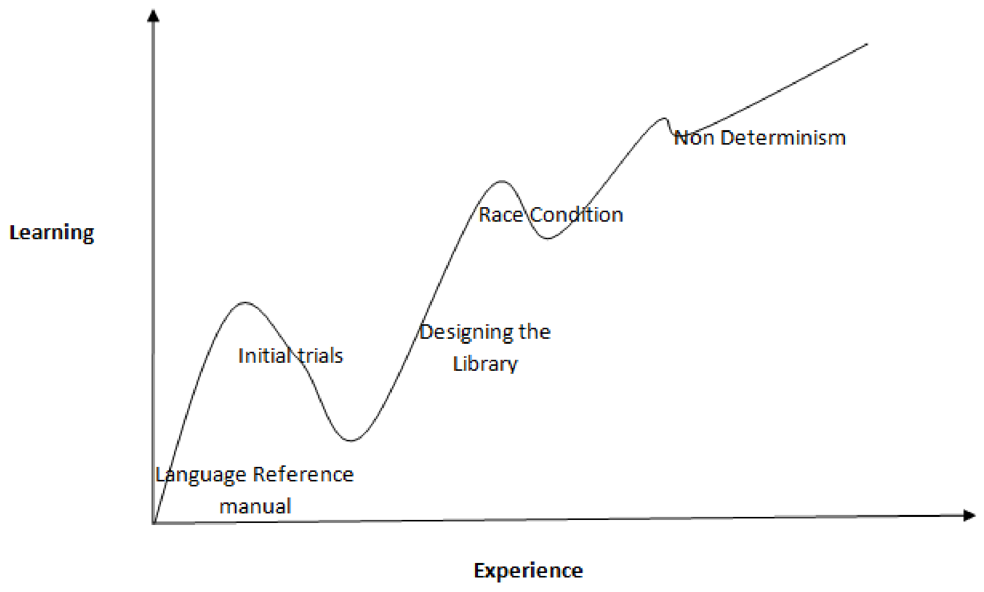


Figure 6.5: Learning Curve

On the positive side, the simulator in SystemC avoided writing the simulation layer.Considering the Simulation layer which was written explicity using java and C++. The layer has five classes : The State Machine which obtains the actions that are to be performed. As the actions could be either urgent or delay, it obtains the detail of the action to be performed from the Urgent Action. The Simulator class then performs these actions of each task till there are no more actions. The interaction between these classes could be explained as follows:

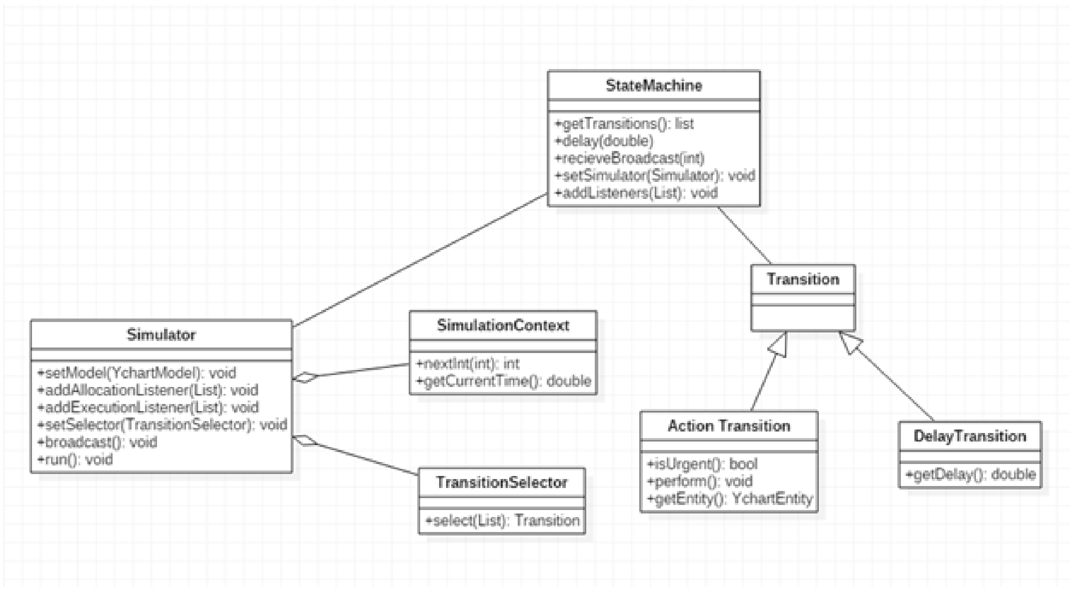

StateMachine
+getTransitions(): list
+delay(double)
+recieveBroadcast(int)
+setSimulator(Simulator): void
+addListeners(List): void
Transition
Simulator
+setModel(YchartModel): void
+addAllocationListener(List): void
+addExecutionListener(List): void
+setSelector(TransitionSelector): void
+broadcast(): void
+run(): void
SimulationContext
+nextInt(int): int
+getCurrentTime(): double
Action Transition
+isUrgent(): bool
+perform(): void
+getEntity(): YchartEntity
DelayTransition
+getDelay(): double
TransitionSelector
+select(List): Transition

Figure 6.6: Simulation Layer in C++

The same simulation layer was implemented in SystemC using the following class diagram:

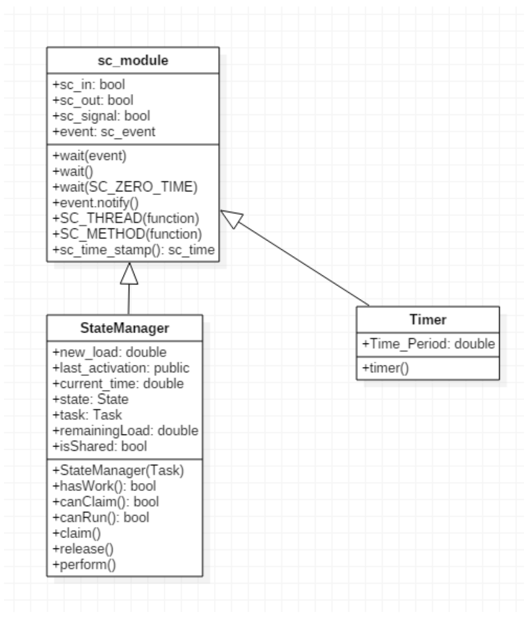
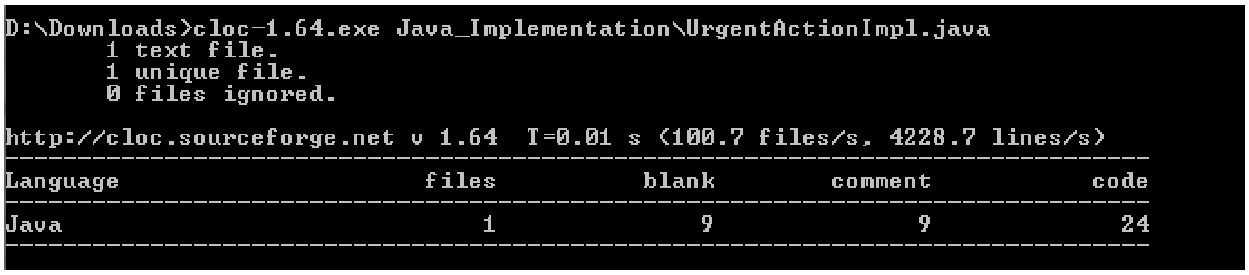


Figure 6.7: Simulation Layer in SystemC

As it can be observed, the number of classes required to model an execution layer in SystemC is half the amount of the number of classes in plain C++/Java.

Lines of Code

Considering the execution layer, the amount of code written in SystemC was lesser when compared with the implementation made in Java/C++. The lines of code were calculated using the software *CLOC*. This software was used to accurately measure the number of lines of code excluding the blanks and comments.



```
D:\Downloads>cloc-1.64.exe Java_Implementation\UrgentActionImpl.java
       1 text file.
       1 unique file.
       0 files ignored.

http://cloc.sourceforge.net v 1.64  T=0.01 s (100.7 files/s, 4228.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                             1              9              9             24
-------------------------------------------------------------------------------
```

Figure 6.8: Line of Code count by *CLOC*

The above figure represents the result obtained from running the CLOC for *UrgentAction-Impl.java* file. Out of 42 lines in the file, only 24 lines of code were written, while there were 9 lines of blank and 9 lines of comment. Similarly, the lines of code were measured for each file and tabulated as follows:

Table 6.1: Java Implementation

Function Name	**Lines of Code**
ODSE.java	79
EngineImpl.java	148
StateManagerImpl.java	164
PriorityActionSelector.java	51
UrgentActionImpl.java	24
DelayActionImpl.java	21
Total	**487**

The above classes were used to perform the simulation as EngineImpl.java runs the simulation creating the State Managers. The StateManagerImpl.java was used to get actions the PriorityActionSelector.java was used to select urgent actions to be performed and the urgent actions were performed through urgentActionImpl.java. The DelayActionImpl.java has the responsibility to obtain the advance in time after a task has finished execution. the ODSE.java is the file in which the simulation starts by calling the *run()* method. These classes are not required for SystemC Implementation as the State Manager was implemented as SystemC module and the SystemC simulation engine was used to perform the simulation. Hence, the lines of code used for SystemC Implementation was as follows:

Table 6.2: SystemC Implementation With ports

Function Name	**Lines of Code**
StateManagerImpl.cpp	280
Total	**280**

Table 6.3: SystemC Implementation Without ports

Function Name	**Lines of Code**
StateManagerImpl.cpp	262
Total	**262**

As it can be observed from the above analysis, the SystemC implementation without ports is having almost two times less lines of code when compared with the Java/C++ implementation.

6.1.4 Memory Usage

With respect to designing in C++, pointers play a very crucial role. That too in case of designing a library with many abstract classes, pointers must be allocated and deallocated with atmost precision in order to avoid memory leaks. ***Visual Leak Detector*** was used in order to detect memory leaks and all the implementation were measured for their performance only after confirming with the report from VLD that none of the implementation has a memory leak. The following measuremnts were taken for the implementations performed through SystemC and C++.

Table 6.4: 2 Task Model - Number of Input Objects: 100,000

Implementation	**Memory Usage (in KB)**
C++ Implementation	400
SystemC Implementation With Ports	544
SystemC Implementation Without Ports	528

Table 6.5: 7 Task Model - Number of Input Objects: 100,000

Implementation	**Memory Usage (in KB)**
C++ Implementation	736
SystemC Implementation With Ports	1496
SystemC Implementation Without Ports	1084

This measurement was performed in order to rule out the memory as a limiting factor for the simulation speed of SystemC. It can be noted from the above results that memory usage by SystemC is minimal and also the increase in memoory from a 2 Task Model to a 7 Task Model is not drastic. Thus, for a system Intel®Core $^{\text{TM}}$ i5-4570 CPU 3.20 GHz, a design with even several thousand tasks could be implemented.

6.2 Conclusions

In this thesis,a library was developed in SystemC and C++ for running models of Data Intensive Applications at a high level of abstraction. A Data Path of a printer case was considered as an example system to use the developed library and perform the simulation. The library is generic enough in simulating data intensive applicaion like the example system (printer data path case) just by elaborating the details of the tasks, resources and their mapping.

- The SystemC simulator was used to perform the activities of both the execution layer and the simulation layer which were explicitly written in Java by [5] and the results were verified by comparing its output trace with the trace obtained from Java implementation.

- After analyzing the domain of SystemC and SystemC AMS, a proof of concept model was implemented and SystemC AMS was left out of the design choice due to its poor simulation speed.

- The Y Chart Layer was built on C++ and made to interact with the simulation engine of SystemC by designing State Manager as SystemC module.

- The State Manager was designed as a SystemC module as it was responsible of task progress from one state to another.

- The State Managers of every task was created as SystemC threads and Timer module was designed to manage the time of the system.

- All the threads were synchronized to events.

- The communication between the State Manager and the Timer module was established through ports and signals. In order to avoid even the communication overhead, the system was also modeled without ports and signals.

6.2.1 Current Findings

1. SystemC was expected to have a similar performance in terms of simulation speed, but it was nine times slower than Java and eighteen time slower than C++. This slowdown was caused mainly due to context switching while running multiple threads parallely.

2. The performance bottleneck for SystemC threads is its wait statement. At every wait statement, these threads suspends with their information saved and retrieved again as they resume. As these wait statements are called multiple times, the speed starts to decrease.

3. The order of evaluation of processes in SystemC is unknown thereby taking much control from the designer. This in turn leads to race condition among the tasks that shares the same computational resources like CPU as the order of evaluation of the threads is unknown. This was overcome by adopting some coding strategies.

4. Though, there is a small degradation in performance due to the communication overhead caused by ports and signals, it is not prominent when compared with a system being used without ports and signals as this kind of system again needs a wait statement or event notification to synchronize the execution of different processes.

5. As the number of objects increases, the speed of the simulation reduces linearly. This is a straight forward relationship as the number of wait statements that are called for increasing number of objects would be more which in turn would cause more context switching causing the slowdown.

6. C++ implementation was the fastest when compared with Java and SystemC.

6.2.2 Recommendation for Future Work

The implementation of a data intensive application in SystemC was researched in this thesis. A future research could focus upon the SystemC at scheduler level and try to parallelize it or optimize it in such a way that the slowdown factor could be reduced. At present SystemC simulation is slower, but it could be improved in future as it is open source and the Open SystemC Initiative is also working ont it. There are other C++ based Discrete Event Simulation tools such as *Adevs* and *PowerDevs*. The application could be developed in these tools and the performance could be checked and benchmarked. POOSL is also another powerful tool which is an Object Oriented Specification Language with a fast discrete event simulation.

Bibliography

[1] Markus Damm. Systemc-ams tutorial. http://www.systemc-ams.org/documents/SystemC_AMS-Tutorial_Damm.ppt. 19

[2] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009. 1

[3] FraunhoferIIS. Systemc ams. http://www.eas.iis.fraunhofer.de/en/business_areas/microelectronic_systems/systemleveldesign/open_source.html. 18

[4] Christoph Grimm, Markus Damm, Jan Haase, Jiong Ou, and Yaseen Zaidi. Refinement of embedded analog/mixed-signal systems with systemc-ams. http://www.systemc-ams.org/documents/date08-tutorial-systemc-ams-1.3.pdf. ix, 19

[5] Martijn Hendriks, Twan Basten, Jacques Verriet, Marco Brass, and Lou Somers. A blueprint for system-level performance modeling of software-intensive embedded systems. *International Journal on Software Tools for Technology Transfer*, pages 1–20, 2014. iii, ix, 3, 4, 5, 6, 7, 22, 23, 24, 29, 64

[6] Open SystemC Initiative. Systemc ams extensions user's guide. http://kona.ee.pitt.edu/socvlsi/lib/exe/fetch.php?media=osci_systemc_ams_users_guide.pdf. ix, 20, 21

[7] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 338–349, July 1997. 2

[8] John Moondanos. Systemc tutorial. https://embedded.eecs.berkeley.edu/research/hsc/class/ee249/lectures/l10-SystemC.pdf. ix, 11

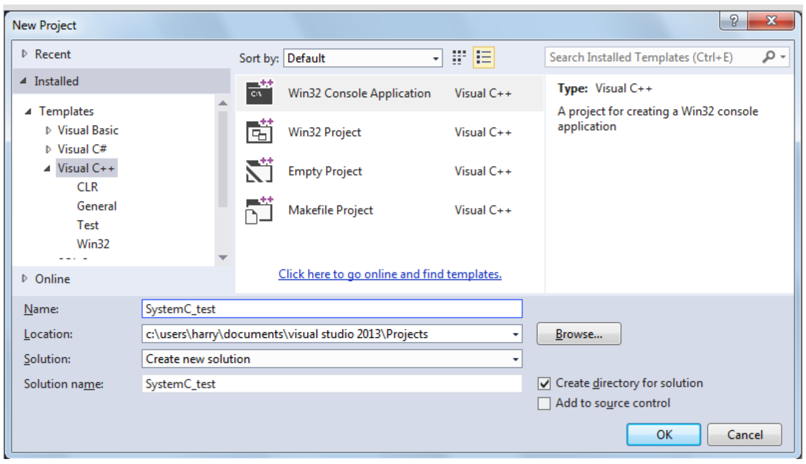
[9] Prateek Sharma. Discrete-event simulation. *International Journal of Scientific and Technology Research Volume 4, Issue 04*, April 2015. ix, 9, 10

[10] Stephen.A.Edwards. Systemc. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/systemc.ppt. 10

Appendix A

Setting Up the SystemC Environment

1. Open Visual Studio and create New project



Type: Visual C++

A project for creating a Win32 console application

Name: SystemC_test

Location: c:\users\harry\documents\visual studio 2013\Projects

Solution: Create new solution

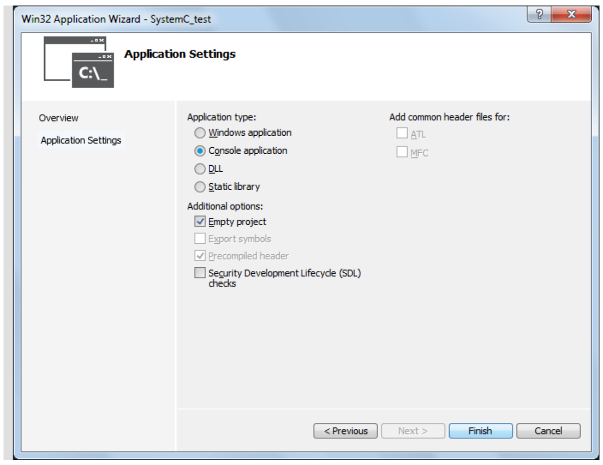
Solution name: SystemC_test

☑ Create directory for solution

☐ Add to source control

Figure A.1: Step 1

2. In that, give a name for the project and select Ok.
3. Now, click **Next** to go to the Application Setting.
4. In **Application Settings**, the following has to be set.


Win32 Application Wizard - SystemC_test

Application Settings

Overview

Application Settings

Application type:
- Windows application
- [✓ Console application]
- DLL
- Static library

Additional options:
- [✓ Empty project]
- Export symbols
- [✓ Precompiled header]
- Security Development Lifecycle (SDL) checks

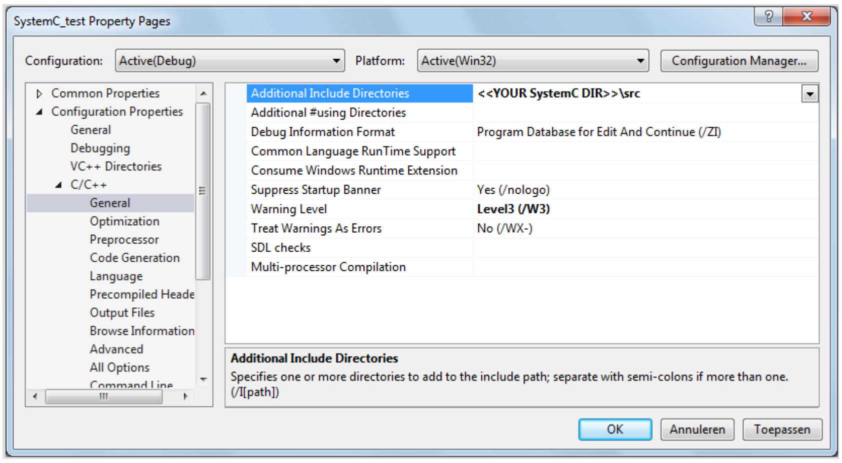
Add common header files for:
- ATL
- MFC

Figure A.2: Step 2

5. Once, the project is created, We need to set some properties which can be done through solution explorer as follows

6. Right click in the project and choose **properties**.

7. In properties box, choose **General** from **C/C++** tab and paste the path of the **src** folder of SystemC there as follows


SystemC_test Property Pages

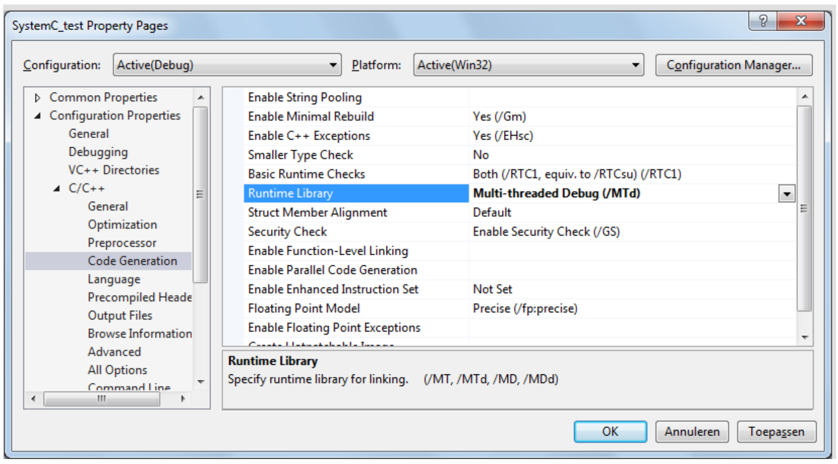
Configuration: Active(Debug) Platform: Active(Win32)

Additional Include Directories	<<YOUR SystemC DIR>>\src
Additional #using Directories	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
SDL checks	
Multi-processor Compilation	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Figure A.3: Step 3

8. Then choose **Multi-threaded Debug** in **Code generation**.



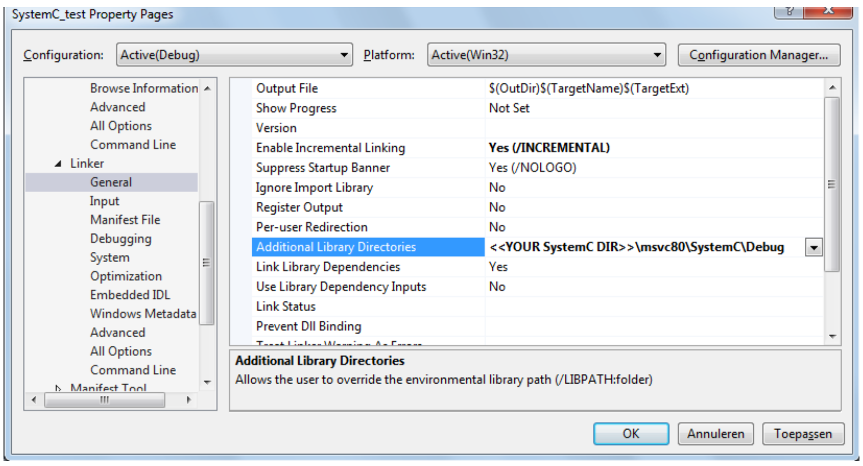
Enable String Pooling	
Enable Minimal Rebuild	Yes (/Gm)
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu) (/RTC1)
Runtime Library	**Multi-threaded Debug (/MTd)**
Struct Member Alignment	Default
Security Check	Enable Security Check (/GS)
Enable Function-Level Linking	
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd)

Figure A.4: Step 4

9. Then add **/vmg** in command line.

10. Now, open the **General** in **Linker** property and add the path where the **Debug** folder of **SystemC** is present.

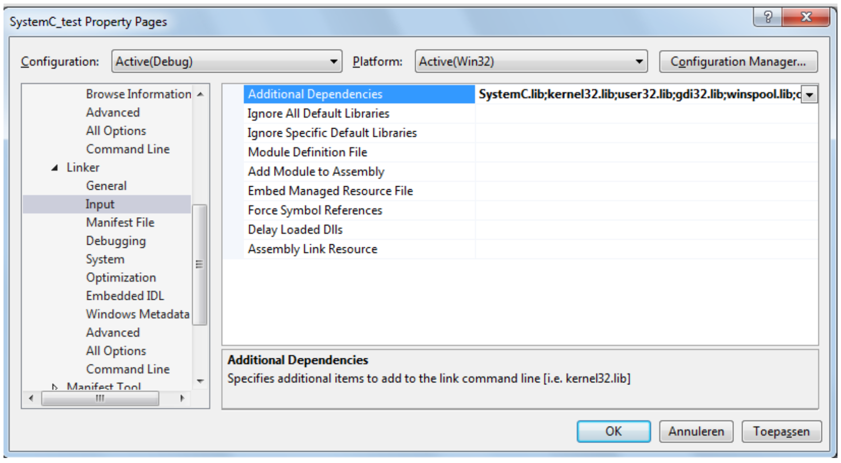


Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**<<YOUR SystemC DIR>>\msvc80\SystemC\Debug**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	

Additional Library Directories
Allows the user to override the environmental library path (/LIBPATH:folder)

Figure A.5: Step 5

11. Now, add the **SystemC.lib** in the **Input** section.


SystemC_test Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)

Additional Dependencies	SystemC.lib;kernel32.lib;user32.lib;gdi32.lib;winspool.lib;c
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line [i.e. kernel32.lib]

Figure A.6: Step 6

12. Now, The SystemC Environment is set, we can start adding the header and source files and start building a project