UNIVERSITY OF CALGARY

An Object Oriented Framework for the Simulation of Network Models

by

Adrian Damian

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

February, 1999

Abstract

Computer simulation is a valuable tool that enables the characterisation of the dynamic behavior of complex systems. The parallel execution of computer simulations aims at minimizing the processing time of the simulation of large and complex physical systems.

Simulation packages are used for reducing the necessary investment required to write sequential or parallel simulations. Typically, they address specific modeling domains and consist of a skeleton program (kernel) and several libraries. The simulation programmer extends the kernel and uses components provided by the accompanying libraries.

This software structure providing an extensible skeleton system is known as object oriented framework (OOF). OOFs have been successfully used in a variety of other fields.

The work presented in this thesis is an example of extending an existing OOF to target a new problem domain. Its main requirements, performance and usability, were the driving force that guided this process. A number of solutions that met these requirements have been identified and their design and implementation are discussed. The advantages and disadvantages associated with different alternatives are studied and trade-off solutions are proposed.

Acknowledgements

Although I cannot possibly acknowledge all of the people who helped me during my study and to whom I am very grateful, I will at least make an attempt.

First, I would like to thank Dr. Brian Unger, my supervisor, for giving me the opportunity and the resources to pursue this work, and for directing my efforts to complete it. Dr. Unger provided tremendous support in bringing this thesis to completion, despite the little time he had available.

My thesis would have not been possible without Xiao Zhonge's help. I am indebted to him for letting me use TasKit and helping me understand and cope with its complexity. Our stimulating and valuable discussions were of great benefit to me.

I owe a huge amount to Rob Simmonds for his continuous help. Rob has been a seemingly limitless source of information about everything from simulation and C++ to style and content of the thesis. I wish Rob had come to Calgary earlier in my stay that he did.

Dr. Mildred Shaw and Dr. Brian Gaines deserve special mention and appreciation for their support, guidance and friendship. I also express gratitude to Dr. Rob Kremer, not only for the time he spent as a member of the examination committee, reading and commenting on this thesis, but also for his patience in answering all my questions about OO, design patterns and frameworks.

I would also like to thank Barbara Dellen, Dr. Frank Maurer, Andy Kremer, Todd Reed and all the other soccer, ski and biking buddies for making the time here so enjoyable. I would also like to thank everybody in the Computer Science Department at University of Calgary, students and office staff for the friendship they extended me during my tenure in Calgary and the enjoyable atmosphere they create.

Back home, in Romania, I want to thank my parents, Ileana and Gheorghe, and my brother, Cosmin, for their endless love and encouragement. They've always trusted and supported my every endeavour, including the choice of pursuing higher education so far away from home.

I've saved the best for last and the best is Dana, my wife. I'm grateful to her for being a source of motivation and unconditional support. She stood by me through all of the inevitable rough times as well as the many enjoyable moments during these years.

Again, the work presented in this thesis would not have been possible without the support of all these people and many others who are not mentioned here. Thank you and I hope I have met the expectations of all who have helped me on my scholarship journey.

To The Damians

Table of Contents

List of Tables

List of Figures

List of abbreviations

API - Application Programmer Interface

ATM - Asynchronous Transmission Mode

ATM-TN - Asynchronous Transmission Mode Traffic and Network Simulator

CCT - Critical Channel Traversing

CMB - Chandy-Misra-Bryant

DES - Discrete Event Simulation

FEL - Future Event List

GVT - Global Virtual Time

LOC - Lines Of Code

LP - Logical Processor

LVT - Local Virtual Time

MIMO - Multiple Inputs Multiple Outputs

MISO - Multiple Inputs Single Output

OO - Object Oriented

OOF - Object Oriented Framework

OOT - Object Oriented Technique

PDES - Parallel Discrete Event Simulation

PP - Physical Processor

RNG - Random Number Generator

SIMO - Single Input Multiple Outputs

SISO - Single Input Single Output

CHAPTER 1: INTRODUCTION

This chapter sets the scene of this thesis by presenting the motivation for this work and briefly describing the concepts used in this research. The goal of the thesis is presented together with a number of objectives to be fulfilled in order that the goal be met. It concludes with a synopsis of the thesis.

1.1 Goal and Motivation

The goal of the thesis is to identify and investigate some of the main issues related to extending an existing discrete event simulation package to target the specific problem domain of *network models* by using object oriented framework techniques. Since *network model* applications tend to be computational intensive, execution performance is a crucial requirement. The emphasis here is placed on issues that concern the design of the application programmer interface (API).

The research in this thesis is concerned with extending an existing software simulation package, SimKit, to address a new problem domain. SimKit has been developed at the University of Calgary (Gomes, Franks, Unger, Xiao, Cleary and Covington, 1995) for implementing fast discrete event simulation applications. The need to extend the current version of SimKit is the result of shifting its main focus towards the domain of *network models* with fixed topology for which very efficient run-time executions of simulations are possible.

Without going into detail (this will be done in the next chapter, Section 2.2.1), a Simkit model is seen as a network of processes that interact with each other by sending and receiving messages. The *network model* presented in this thesis extends this model by

requiring that process interactions can be described by a static graph. The definition of *network models* used here is similar to those used for distributed systems:

> *A network model is a model in which component interactions can be described by a weighted directed communication graph whose nodes are the processes, and where an arc from process i to process j denotes that i can send messages directly to j but not vice versa. Weight on each arc in the graph represents the delay on that arc, that is, the minimum difference between the receiving and the sending time of a message*[1].

This thesis presents a novel approach to analysing a simulation package in general and SimKit in particular, by using an object oriented framework technique. Relevant concepts related to discrete event simulations (DES), the SimKit simulation package and object oriented frameworks (OOF) are briefly introduced in the subsequent sections.

1.1.1 Discrete Event Simulation

Computer simulation is the discipline that allows people to study complex dynamics of physical systems by designing a model of the system, executing the model on a computer and analysing the execution output. One of the key strengths of simulation is that it enables the study of a system over time. An approach for controlling the time advance in a simulation is to step the model into the future only at discrete, possibly random points in time when an event that could change the state of the system occurs. The nature of this type of simulation, known as discrete event simulation, enables a model to be more quickly evaluated since only the significant points in time when something happens in the system are considered.

DES is used in solving problems in a variety of domains such as service industries, manufacturing, telecommunications (Unger, Xiao and Cleary, 1999), to name a few. Computer simulations of such applications are becoming larger and more complicated as

---

[1] The terms message and event are used interchangeably

people need to model more complex systems. Although computer power continually increases, simulations of such large systems are still time consuming. Parallel discrete event simulation (PDES) has as its main goal the reduction of execution time of a DES application by executing it concurrently on multiprocessor computers. A large body of research has been conducted in this direction and a number of approaches for solving the main issues related to PDES such as synchronisation, scheduling, memory management, partitioning and load balancing, have been proposed. Although encouraging results have been obtained with different approaches, they are highly dependent on the application characteristics and none proves to be a universal solution that can be successfully applied to any type of simulation problem.

Carrying out a DES experiment on a computer can be done in different ways: using an available simulator or implementing the model using a special simulation language or general purpose programming language. The general programming language alternative is the most flexible one but writing a simulation program from scratch may be a lengthy undertaking. With the emergence of object oriented techniques this difficulty can be overcome. Object-oriented techniques (OOT) (Booch, 1994) have gained popularity in recent years and the simulation community was one of the first to use them with the introduction of the Simula programming language (Pooley, 1987). The power of object oriented techniques lie in their ability to produce "modular" code (known as classes) that can be "easily" modified and reused (Shewchuk and Chang, 1991). Taking advantage of the benefits of the object-oriented paradigm by using libraries and "skeleton programs" can reduce the development effort.

There are a large number of simulation packages designed to support the implementation of DES applications using a general programming language. Most of these packages consist of a "skeleton program" and a number of libraries. Libraries in a package capture the most common components found in many DES applications while the skeleton program implements the basic mechanism of simulation that is customised by each application according to its own behavior. This software structure is not specific only to

the simulation world but to many other application domains and it is known as object-oriented frameworks (OOF).

1.1.1.1 SimKit

SimKit is a simulation package that implements a simple PDES mechanism. It has been designed to facilitate building high performance sequential and parallel DES applications using the C++ programming language[2].

The main component of the SimKit package is the "skeleton program" that captures a classical PDES mechanism common to all its applications. Using inheritance and other OOT, this skeleton is customized by the user to implement the particular behaviors of different models.

The main advantages of SimKit include:

- *Simple API.* The small number of components that are used by an application make SimKit easy to learn and understand.

- *Efficiency.* As in any PDES package, high performance is the driving force in SimKit development. Several optimized synchronization kernels can be used with SimKit to facilitate good execution performance for a range of DES applications.

- *Portability.* Applications using SimKit can be executed on a variety of sequential and parallel platforms from personal computers to supercomputers.

These benefits allowed SimKit to be successfully used in developing different PDES applications and this has been an incentive for continually improving it. Most of the changes aim at improving its performance. As mentioned earlier, no available PDES algorithm is universally suitable in all situations. However, different algorithms can be implemented and optimised to take advantage of the particular properties of specific types of applications. Other cathegories of changes are designed to make SimKit easier to use.

This includes different libraries for components that can be reused with applications and a number of new features for modeling and development processes.

The goal of this thesis is to analyse and describe such a step in SimKit's evolution. Most of the changes presented here are the result of narrowing down the application domain from general DES applications to *network models*. This new problem domain allows the introduction of several optimizations in the kernel. Other changes refer to new modeling features and the implementation of new component libraries.

Object oriented frameworks concepts and techniques, described in the next section, are used for achieving this goal.

1.1.2 Object Oriented Frameworks

Reusing the products of the software development process has been a goal for software engineers for many years. The object-oriented paradigm has features that encourage reuse especially at the level of software components (objects). Furthermore, object oriented frameworks (OOF) is a promising technique that enables not only code reuse but also design and analysis reuse (Johnson and Russo, 1991).

An OOF, or simply framework, is essentially the design of a set of objects that collaborate to carry out a set of responsibilities in a particular problem domain (family of applications). It is typically delivered as a collection of interdependent abstract classes possibly together with a set of concrete classes. The programmer's responsibility is to specialise the abstract classes and instantiate the provided concrete classes as objects to create a desired application. In this way, an application becomes a new member of the family.

A successful framework has the following attributes (Adair, 1995a):

---

[2] There is also a Java version of the sequential SimKit package but this is not discussed in this thesis

- Complete - the framework provides default implementation and build-in functionality where possible in order to support features needed by the users;
- Flexible - abstractions can be used for different applications in the given problem domain;
- Extensible - provide hooks that enable the framework to be customized for the specific needs of an application;
- Understandable - the framework is well documented so that its features are easy to use correctly.

However, developing a successful framework is a difficult undertaking and therefore it usually comes as the result of an iterative process consisting of several phases (Johnson, 1992). A framework often is inheritance-based at the beginning of its life-cycle, since the application domain is not sufficiently well understood to make it possible to parameterize its behaviour and evolves towards a "black-box" type of framework as more information about their application domain is gained.

Designing and developing high quality frameworks is more difficult than developing conventional software for the same problem (Gamma, Helm, Johnson and Vlissides, 1995) (Fayad and Schmidt, 1997). Besides the skills that sometimes only programming experts have, a successful design requires also very good understanding of the problem domain. Other challenges of developing frameworks refer to maintenance, validation, defect removal, efficiency etc.

Nevertheless, the benefits of a successful framework over its lifetime might well outdo these weaknesses. The best approach in developing successful frameworks is to understand the framework concepts, its components, strengths and weaknesses. By knowing the weaknesses of a framework they can be addressed and minimized while its strengths can be fully exploited.

1.1.3 Upgrading a Simulation Tool Using Frameworks Techniques

DES is one field that is very suitable to the framework approach. The basic DES simulation mechanisms are captured in simple algorithms. Therefore these algorithms can be implemented by frameworks to be reused in different applications. This is exactly what most DES simulation packages do.

Although object oriented frameworks have been discussed for several years and most of the DES packages have a framework structure and characteristics, there is no research to specifically study the applicability of object-oriented frameworks to the DES domain. The author believes that the knowledge and experience gained from developing OOF in different application domains can be used towards improving the design and usability of frameworks applied in the DES domain, referred shortly as DES frameworks.

Filling the gap between these two domains is expected to bring some advantages for DES frameworks:

- improve DES framework design.
- understand and exploit the strengths of DES frameworks
- identify pitfalls so their effect can be reduced in DES frameworks
- increase DES framework usability
- suggest improvements to DES frameworks
- suggest future research directions in DES frameworks

1.2 Objectives

This thesis has the following objectives:

1. To study discrete event simulation, sequential and parallel, and to frame an in depth understanding of the main issues in this field.

2. To study object-oriented frameworks and to get insight into this field.

3. To study an existing discrete event simulation framework, SimKit. This includes identifying its initial requirements and understanding its mechanism and implementation.

4. To identify directions in which SimKit could be extended and to choose an approach for performing these changes.

5. To analyse some of the issues related to the implementation of the changes in the extended SimKit and to discuss alternatives for addressing these issues.

6. To draw some conclusions from this process in a form of "lessons learned" and to propose further development and research directions based on the experience and insights gained.

1.3 Thesis Structure

Chapter 2 addresses Objective 1 by giving a brief description of the current state of the art in computer simulation with the main focus on discrete event simulations. At the beginning it presents an overview of the modeling and simulation concepts. Then the focus is shifted towards describing different alternatives for modeling a DES problem. The presentation continues with an introduction to several existing DES algorithms both for sequential and parallel execution.

Chapter 3 addresses Objective 2 by introducing the concept of object-oriented frameworks as techniques that enable software reuse at large scales. This chapter also provides background knowledge on object-oriented techniques. The description of frameworks includes a presentation of their components, their importance, some of the strengths and pitfalls they might cause as well as a discussion of framework evolution, problems and approaches related to their evolution process.

Chapter 4 addresses both Objective 3 and Objective 4. The focus of this chapter is SimKit, an existing simulation package intended for very efficient, sequential and parallel execution of discrete event simulation applications. A description of SimKit's implementation and API is provided. The need and directions for future developments in SimKit are determined. SimKit is identified as being an OOF and the approach of using three simulation models in order to gain domain knowledge suggested by OOF practitioners is chosen for extending it.

Chapter 5 addresses Objective 5 by analyzing the new functionality that can be added to SimKit in order to meet the needs identified in the previous chapter. These new features are analysed and possible approaches are suggested.

Chapter 6 addresses Objective 6 by summarising and discussing the main points of the thesis. It includes an evaluation of the proposed changes in Simkit and limitations of this evaluation. The overall work presented in this thesis is also evaluated and suggestions for future research directions are proposed.

1.4 Summary

This chapter presented the goal of the thesis and a number of objectives that have to be fulfilled in order to meet this goal. The two main fields that the thesis addresses are discrete event simulation and object oriented frameworks. A brief introduction of their main concepts as well as possible benefits of combining the knowledge and experience of the two research areas are described.

CHAPTER 2: DISCRETE EVENT SIMULATION

This chapter surveys the simulation field, especially discrete event simulation. There are different techniques that can be used for the analysis and synthesis of a complex system such as statistical techniques, laboratory experimentation, hardware prototyping and modeling using analytical and simulation approaches. Among them, computer simulation is often the tool of choice due to its ability to experimentally explore the dynamics of complex real world systems at a low cost.

Discrete Event Simulation (DES) refers to modeling of systems in which the state changes only at discrete points in time, as opposed to continuously over time. DES is used in a variety of domains ranging from manufacturing to healthcare, from telecommunication to economics.

As the complexity of systems under investigation increase, so do the users' expectations for better simulators. The main requirements for simulation modeling are: efficiency (speed and memory), accuracy, and ease of use. As a result, the research activity in this field is oriented towards discovering new algorithms and application programmer interfaces (APIs) that successfully meet these requirements. This chapter surveys some of the main directions in DES research, some of the results and challenges, with the focus on the performance aspect. It is organized as follows:

Section 2.1 briefly identifies the main phases of a simulation study as well as its importance. Section 2.2 defines the two principal types of simulations: continuous and discrete as well as four modeling views of the world: *event driven, process oriented, activity scanning and the logical process view*. Section 2.3 emphasises the need for

efficient simulators and identifies the parallel execution of a simulation as a possible solution for this problem. Parallel discrete event simulation (PDES) becomes the focus of section 2.4. Two well-known classical PDES algorithms, Chandy-Misra-Bryant (CMB) and TimeWarp, are presented here. A new scheduling mechanism, Critical Channel Traversing (CCT), aimed at the simulation of *network models* with sparse connectivity and high message density on shared-memory multi-processors computers is also described.

2.1 Modeling and Simulation

Simulation enables the characterisation of the dynamic behavior of a new system or a modification to an existing one, before committing to a course of action. It allows the evaluation of the performance of the system under different configurations of interest, operating scenarios and over selected periods of time. Simulation is widely used for system analysis, performance evaluation, sensitivity and cost effectiveness tests, forecasting, training and decision making.

A simulation study is an iterative process typically composed of several steps grouped in three main categories (Figure 1)(Maria, 1997):

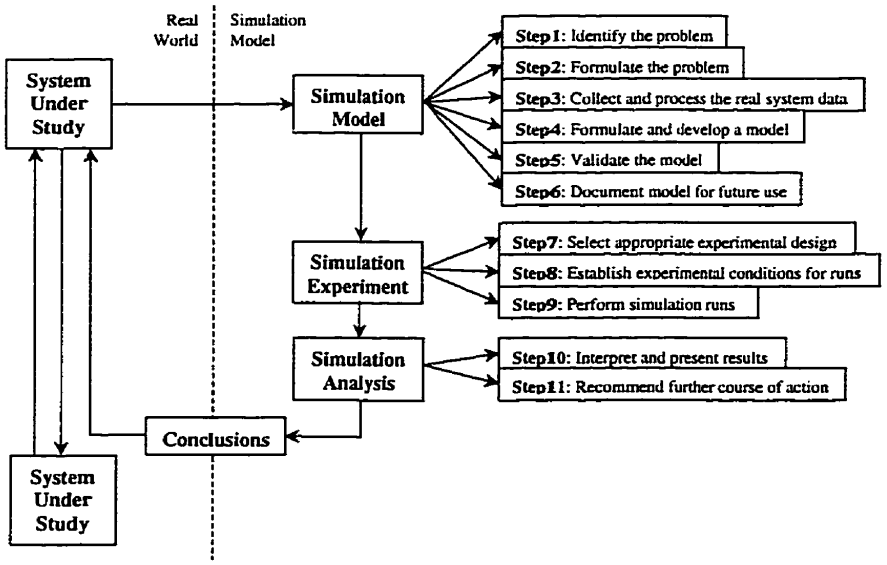


Figure 1 Steps in a simulation study

1. ***Simulation Model.*** A model is an abstract representation similar to, but usually simpler than the physical system it describes. It contains structural, logical or mathematical relationships that describe the system to be studied. The model of a system should represent a close approximation of the real system. It should not be excessively complex, making it impossible to understand and experiment with nor so simple that it introduces errors and does not accurately reflect the actual behavior of the system. There is always a trade-off as to what level of detail is included in the model.

   The process of determining the correctness of the model is known as validation. Validation involves comparing the model's behavior under known input conditions with the behavior of the real system.

2. ***Simulation Experiment.*** A simulation experiment involves a series of tests in which the results of meaningful changes in the input variables of a simulation model are studied. The main objective is to identify the reasons for changes in the system's behavior.

3. ***Simulation Analysis.*** The results of a simulation have to be interpreted in order to draw conclusions and recommend further courses of action. The main difficulty in this phase is to distinguish the results due to meaningful changes in the input variables from those caused by a random occurrence. The latter can lead to misinterpretations of the meanings of the simulation results.

There are different alternatives for modeling a problem. Usually, the *physical system* is decomposed into a collection of *physical processes*. An *entity* is the representation of a *physical process* in the *logical system* that models the *physical system*. A *property* of a *physical process* is represented by an *attribute* of the *entity* that describes it. As a result, all the *attributes* of all *entities* in the model *describe* the state of the *physical system* at a given instance of the simulation time. On a computer, the *logical system* is implemented by a *simulation program*, a type of *entity* by one or more *entity classes* (if an OO language is used) and *attributes* by *class variables*. One particular *entity* in the logical system is

modeled by an instance (object) of its corresponding entity class. The terminology corresponding to each of these three cases is summarized in Table 1.

Table 1 The corresponding terminology for the physical system, its logical model and the implementation of the model on a computer

Real world system	**Model**	**Computer**
Physical system	Logical system	Simulation program
Physical process	Entity	Entity object
Physical process' property	Entity's attribute	Entity class' variable

Computer simulation is the discipline of executing a model on a computer. Hereafter in this thesis, simulation refers to computer simulation.

2.2 Discrete Event Simulation (DES)

Real world systems can be categorized as *continuous* or *discrete,* depending on the state variation behavior prominent in the system . Consequently, the two types of simulation paradigms that describe these two views have different characteristics:

- ***Continuous Simulation*** models systems that change their state continuously over time. It typically describes the behavior of a system using sets of equations, which are solved numerically with respect to time. The continuous simulation program advances by the constant increment of time previously chosen for the time step and recalculates all equations that describe the model. Examples of problems suitable for this approach are fluid-flow or hydraulics problems, geophysics models, etc.

- ***Discrete Event Simulation*** refers to the modeling of systems in which the central assumption is that the system's state changes instantaneously at discrete, possibly random points in time. The rest of the thesis deals exclusively with DES.

In DES the state of a system changes at discrete points in time in response to certain discrete events. An event models an action in the physical system that occurs at an instant in time. Simulation time increases as a monotonically non-decreasing function. Causality refers to the fundamental principle of real world systems according to which the future cannot affect the past. This imposes a chronological sequence in state transitions and consequently the order of event execution during a simulation. Violations of this principle cause errors in simulation referred to as *causality errors*.

2.2.1 Mechanisms for Describing the Logic in DES

There are four main ways for describing the logic in DES. These modeling approaches or views are: *event-driven, activity scanning, process-oriented* and *logical process*. Each approach offers a different way to describe the logic of a simulation problem but the results of modeling a system using any of these modeling views should be the same. The differences lie in the ease in which they can be understood and implemented and in the efficiency of their executions.

Figure 2 illustrates the first three modeling views: event, activity and process for modeling the lifecycle of a machine (Ball, 1998).

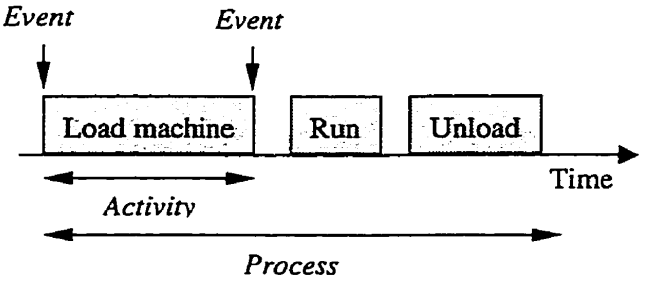


Figure 2 Modeling views

The main characteristic of the *event-driven view* is that any state change in the model occurs by explicit scheduling of an event. A discrete event is something that occurs at an instant of time. The start of machine loading, for example, is a discrete event because occurs at an instant of time. Activities such as loading a machine (Figure 2) are not discrete events because they have a time duration. However, they can be modeled as a pair of

events that signals the start and the end of the activity. The duration activities are therefore modeled as the difference between the times associated with the events marking the beginning and the end of each activity. Routines are implemented to describe changes in the state of the model as a result of different types of events (e.g. from idle state to load or run state etc.).

In the *activity-scanning view* the logical system is composed of a number of activities. Activities describe a duration, e.g. machine loading, machine run etc. (Figure 2). The occurrence of a state change in the system is chosen based on both the scheduled times for the next activity and a condition testing. This approach offers a relatively simple way to model a problem. To date, however, its main disadvantage, inefficiency, is a big impediment making this approach unpopular.

With the *process oriented* view a logical system consists of a number of active entities whose behaviour is of interest. Collections of events or activities are joined together to describe the lifecycle of an entity, such as the machine in Figure 2. A set of process routines describes the actions of these entities. A process routine, unlike event routines in event-driven view, may allow the passage of simulated time. Therefore, they usually require special mechanisms called co-routines for interrupting and suspending the execution of a process routine. Its execution resumes at a later simulated time under the control of an internal event scheduler. This mechanism allows an entity to "spontaneously wake up" and begin new computations without first receiving a message. Process oriented logic simplifies large models by allowing many aspects of an entity's behaviour to be described in one method.

The *logical process (LP) view* is the dominant modeling framework in the parallel discrete event simulation (PDES) community. The *LP view* adds a *process oriented* modeling representation to the *event view* in that the physical system is decomposed into disjoint physical processes described by separate *logical processes* or for short *LPs* (*LP* is the equivalent of an entity). However, unlike in the *process oriented view*, the simulation time passes only with the arrival of events scheduled by the *LPs* in the system. Any state change

in an *LP* is caused only when an event occurs (is received) at that LP. There are no shared attributes among *LPs* since physical processes they described are chosen to be disjoint. Scheduling events is the only mechanism that allows an *LP* to change or to read the state of another LP. This mechanism permits minimal processor synchronisation, making the *LP* view particularly suitable for PDES.

2.3 Parallel Execution of a Simulation

In a traditional event-driven sequential DES, events are scheduled in a central chronological ordered queue usually referred to as the Future Event List (FEL). The FEL at any point in simulation time contains those events that have been scheduled to occur. The simulation time advances when the event with the lowest timestamp is removed from FEL and processed by its corresponding event handler. The execution of an event may lead to changes in the system state and may cause new events to be scheduled in the future. A simulation terminates when either the FEL becomes empty or the system reaches a specified simulation time or condition. This mechanism guarantees that the causality constraint cannot be violated because all the events are always processed in a chronological order.

DES is extensively used for simulating different complex systems in telecommunication, manufacturing, training, traffic, engineering, combat, to name a few. These models are increasingly large as people try to study more complex problems. A simple simulation experiment of a telecommunication network, for instance, may require 10^{12} events to be processed (Unger et al., 1999). As a result, execution time for such models increases dramatically and new efficient approaches are needed.

One solution that has attracted a considerable amount of interest in the last two decades is to run a simulation program concurrently on multiple processors. There are a number of alternatives that are all driven by the same common goal: to accelerate or speedup the execution of a simulation.

Probably the simplest alternative is to execute independent, sequential simulation programs or trials on different processors. This approach is less suitable for studies with a low number of trials or when the results from one trial are used to determine the settings for the next ones. Another alternative is to use support function parallelism. This implies running in parallel only specific sequential simulation functions such as event list manipulation and random number generator. All the other functions are executed sequentially, which means that the simulation runs only partly in parallel. The method, therefore, offers only a limited amount of speedup.

Finally, Parallel Discrete Event Simulation (PDES) is the approach considered in this thesis. The main idea is to decompose the simulation application into a set of concurrently executing processes and run them on different processors. This approach attempts to exploit the parallelism innate in the real world system. Therefore it is considered as the most promising alternative and it is the focus of the next subsections of this chapter.

2.4 Parallel Discrete Event Simulation (PDES)

The *Logical process* (LP) view is the preferred mechanism for describing the logic in PDES. *LPs* are assigned and executed concurrently on different processors. This is facilitated by the fact that *LPs* in the logical system are disjoint; *LPs* do not share attributes with other *LPs* and therefore it requires minimal processor synchronization.

LPs communicate with each other by sending timestamped events. The timestamp of an event represents the time when the event is scheduled to occur at the destination. Because *LPs* are executed concurrently, there is no global simulation time. Instead, each *LP* maintains its own logical clock denoted by Local Virtual Time (LVT). The LVT of an *LP* advances to represent the timestamp of the last message received.

This mechanism allows concurrent asynchronous execution by running *LPs* on different processors in the system. The main challenge with this approach is to maintain certain sequencing constrains that dictate the order in which computations must be executed relative to each other. More specifically, even if *LPs* are concurrently executed on different

processors, the algorithm has to ensure that no causality errors ever occur. The following rule ensures that the sequencing constrains are maintained:

> "Local Causality Constraint - A discrete event simulation, consisting *of logical processes* (*LPs*) that interact exclusively by exchanging timestamped messages, obeys the local causality constraint if and only if each *LP* processes events in nondecreasing timestamp order." (Fujimoto, 1990. p. 32)

This constraint, while guaranteeing that no causality errors occur, can be unnecessarily restrictive. For example, two events within a single *LP* that are independent of each other can be processed in an arbitrary order without changing the result. This implies that even if they do not obey the above rule they do not introduce any causality errors.

Synchronization of *LPs* is particularly difficult in PDES because of the nature of sequencing constraints imposed on this type of application. Unlike other areas of parallel computation that have been highly successful (vector operations on large matrices of data - for example) in PDES these constraints are complex and highly data-dependent. As Fujimoto states, "the dynamic nature of the PDES problem is the principal reason that a general solution has been elusive" (1990, p. 32).

The synchronization mechanisms in PDES traditionally fall in two main classes: conservative and optimistic. The next two sections briefly describe the two approaches. The Critical Channel Transversal algorithm presented later provides an efficient way of scheduling events in a conservative fashion.

2.4.1 Conservative Approach

The first distributed simulation mechanisms, developed almost two decades ago, were based on conservative approaches. They have been designed independently by Chandy and Misra (1979) and Bryant (1977).

The main idea in conservative schemes is to strictly avoid the possibility of any causality error occurring. An *LP* cannot process an event unless it is safe to do so, that is, no

messages with an earlier timestamp can arrive. The safe time of an *LP* can be determined assuming that links that indicate which *LP* may communicate with which other *LPs* are statically specified at the beginning of the experiment (the topology of the *LPs*) and messages arrive on each incoming link in timestamp order. The *LP* is blocked until it is safe to process its next event. If appropriate precautions are not taken this can cause deadlock situations in which all *LPs* are waiting for their neighbour *LPs* to send messages.

Either a deadlock avoidance mechanism or a detection and recovery approach may be used to mitigate this problem. The mechanism commonly used to avoid deadlock is to send NULL messages. NULL messages are used only for synchronization purposes and therefore they do not correspond to any activity in the model. A NULL message is sent to all outputs when no more events can be executed and contains a timestamp that is a lower bound on the timestamp of any future events to be sent on that link. Lookahead is the timestamp increment of a NULL message and "refers to the ability to predict what will happen, or more important, what will not happen, in the simulation future" (Fujimoto, 1990, p. 36). It has a great impact on the performance of the application. With this mechanism, cycles[3] of zero timestamp increment are prohibited as they could lead to deadlock.

Deadlock detection and recovery mechanisms do not use NULL messages. Instead, they use a mechanism for deadlock detection and yet another one for breaking the deadlock. Processing the *LP* with the smallest timestamp is always safe and it can break the deadlock. Therefore this mechanism can be used to detect and break only from global deadlocks.

There are many variations of the Chandy-Misra-Bryant conservative mechanisms Fujimoto (1990), none of them achieve best results for all types of problems. Although the conservative algorithm has a relatively low system overhead, its performance is subject to

---

[3] In the LP interaction graph, any event generated by an LP in a cycle could effect the generation of an event that will subsequently arrive at the same LP

a number of factors such as: the message density, load balancing (how *LPs* are assigned to processors so that the execution load is evenly distributed), lookahead in cycles. Nevertheless, conservative methods offer potential for obtaining good speedups particularly for applications in which there is uniform message density across all *channels* and there exists good lookahead.

2.4.2 Optimistic Approach

Unlike the conservative mechanism, the optimistic approach allows causality errors to occur. Each *LP* executes aggressively according to its own local virtual time (LVT) independent of other *LPs*' LVTs. The optimistic assumption that is made is that no events with timestamps earlier than the LP's local time will ever arrive. However, when such an event occurs, a causality error is detected and a rollback mechanism is invoked to undo the effects of all events that have been processed prematurely. Events are then re-executed in order thus obeying the Local Causality Constraint (section 2.4). Optimistic schemes allow the simulation to exploit parallelism in those situations in which causality errors may occur but in fact do not.

Jefferson's TimeWarp algorithm (1985) was the first optimistic protocol. In TimeWarp, each *LP* saves old values of its attributes and maintains a history of past states. When an event with a timestamp lower than the LVT, known as *straggler*, is received it triggers a rollback. The *LP* restores its previous state and cancels the effect of the events that have been prematurely sent out to other *LPs* by sending negative events named anti-messages to annihilate the originals. If the original event, referred to as the positive message, has already been processed at the destination, then the destination *LP* must also be rolled back. Thus, an *LP* rolls back whenever it receives a straggler message or an anti-message and can cause further rollbacks at the destination resulting in a cascade effect. In spite of this, it can be shown that this mechanism always makes progress under some mild constraints (Jefferson, 1985).

In Time Warp protocol, the smallest timestamp among all unprocessed event messages is called Global Virtual Time (GVT). No event with timestamp smaller than GVT will ever

be rolled back, so the saved states can be discarded for such events. The process of reclaiming memory is referred to as *fossil collection*.

While the optimistic mechanism has a great potential of fully exploiting the parallelism available in an application there is a high overhead associated with operations such as state saving, rollbacks, GVT computations, event cancellation. The overhead cost is particularly noticeable for fine grain simulations. This type of simulation is characterised by low average time spent executing event-related procedures which makes the system overhead account for a large part of the application execution time. Research has been conducted to reduce the overhead in optimistic schemes by optimising some of the operations. Different algorithms for event cancellation (Fujimoto, 1990), state saving (Gomes, 1996) or GVT calculation (Gomes, 1993) have been proposed. Again, none of these alternatives offers a general applicable solution for all types of problems and the results can vary dramatically even for small modifications within the same application.

Optimistic mechanisms such as Time Warp have proven to be, in general, more robust that their conservative counterparts. They have a good chance of success assuming that these overheads can be kept to a manageable level. This is case for large granularity models in which the amount of computation per event is large and amortises the overhead computations. However, as Fujimoto points out (Fujimoto, 1993), increasing event granularity may be difficult, depending on the application.

2.4.3 The Critical Channel Traversing (CCT) Approach

The Critical Channel Traversing (CCT) is a new extension of the Chandy-Misra-Bryant (CMB) conservative synchronisation algorithm. It is aimed at the very fast simulation of low-granularity *network models* on shared-memory multi-processor computers (Xiao, Unger, Simmonds and Cleary, 1999).

The CCT algorithm is designed for *network models*. *LPs* are connected to each other through *channels* through which events can be sent. The only exception to this rule is when an *LP* sends an event to itself (self event). Therefore, *LPs* and *channels* in the logical

system represent nodes and respectively arcs in the interaction graph of the application. Based on the connectivity graph, *LPs* are grouped into *tasks* and scheduled as a single unit. Tasks have no equivalent in the model; instead their existence is for the purpose of efficiency in scheduling.

The CCT approach tries to overcome some of the performance problems that the optimistic and classical conservative algorithms face with low granularity network simulations by addressing three key issues:

- ***Per-event system overhead.*** As mentioned earlier, this is relatively lower in conservative schemes than in optimistic ones (Fujimoto, 1990), (Unger et al., 1999).

- ***Dynamic load balancing.*** Load balancing or allocating *LPs* in the model to the processors is one of the big issues in PDES. Usually, a simulation model consists of a large number of *LPs*, far more than the number of available processors. Achieving good performance requires careful load balancing, otherwise overloaded processors will be slower than the others delaying the execution. This is the reason, the performance of both conservative and optimistic schemes greatly depend on this factor. The load balancing, a difficult issue for all conventional parallel methods becomes even more difficult with large *network models* due to the dynamic nature of traffic being modeled.

- ***Program cache locality.*** Classical parallel algorithms use different schemes for scheduling *LPs*. Some of these schemes obtain poor cache behaviour due to the small number of events that each *LP* is capable to execute when it is scheduled.

These are believed to be the main obstacles that limit the performances of previous PDES algorithms. The CCT algorithm is the result of intensive experiments with networks with static *LP* interconnectivity. It exploits the characteristics of such an application and addresses the three issues mentioned above in an original way:

"Performance is achieved through a multi-level scheduling scheme that supports the scheduling of large grains of computation even with low-granularity events. Performance is also enhanced by supporting good cache behaviour and automatic load balancing" (Xiao et al., 1999).

How CCT addresses these three key issues is described in the following sections:

2.4.3.1 System Overhead

In CCT, per-event overhead is minimized by using the conservative synchronisation approach. Moreover, the original conservative algorithm is highly optimised, which ensures an even lower event overhead. One of the main requirements introduced by this paradigm is that the model interaction graph be statically defined at the beginning of the simulation. Static *channels* are used to connect the *LPs* in the model. All the events in the application except *self-events* that are sent by an *LP* to itself are sent exclusively through *channels*.

2.4.3.2 Load Balancing

In general, the load balancing in PDES can be done statically or dynamically. The first alternative implies that *LPs* are assigned to processors only at the beginning before the simulation experiment starts. The workload has to be evenly distributed among all the processors to avoid situations in which overloaded processors slow down the entire execution. This is not trivial since it is difficult to estimate or measure accurate information about the execution load on each processor. Moreover, as parameters of the simulation change, the workload on processors changes as well, so that a separate partitioning has to be done for each experiment. This is not a feasible solution especially for large models with hundreds of *LPs* and therefore a dynamic load balancing scheme is needed in such situations.

A naïve approach of dynamic load balancing is to have a single centralised FEL that the next available processor takes the next event from and processes it according to the code implemented by its handler. This scheme, although ensuring an almost perfect balancing of

the execution load yields poor performance because of contention accessing the shared queue.

CCT uses the same idea of having a centralized[4] queue for realizing the dynamic load balancing. The main difference is that events in the shared queue are replaced by *tasks*. Tasks are used for grouping *LPs* that have a high dependence on one another. High dependence is defined as:

> "...if an event is executed by one *LP* in a task, it is likely that the execution of this event will lead to the generation of an event for another *LP* in the same task." (Xiao et al., 1999)

As a result, each processor takes the next *task* (instead of the next event) from the queue and executes it. The granularity is increased so each processor accesses the queue less frequently and the contention problem is greatly reduced. Also, the load balancing process is reduced since *LPs* are allocated to specific *tasks*. This process is simple compared to the static partitioning of *LPs* to processors mentioned earlier because it is based only on simple topological and *task* type information. It is not even necessary to measure or estimate the workload on each task.

2.4.3.3 Cache Locality

By executing highly dependent *LPs* within *tasks*, the cache locality also improves. There are currently two types of *tasks* in CCT: *pipe-tasks*, used for grouping *LPs* that are connected in pipeline configurations, and a *cluster task* type that is used for groups of *LPs* that appear in low lookahead cycles. These types of *tasks* handle the *LP* scheduling differently. Each *cluster task* implements its own event list that stores all the events that have been scheduled and not processed yet at the member *LPs*. The execution within a *cluster task* is similar to that on any sequential DES: each event is removed from the list

---

[4] The possibility of a sharing queue is also currently under investigation.

and executed at its destination LP. The cache locality might improve in this case because the processor that executes the *task* works only with a subset of all the *LPs* in the model.

The cache behaviour can however improve further with *pipe-tasks*. The configurations on these types of *tasks* allow the *LPs* to be scheduled one after another starting at the beginning of the pipeline to the end. There is no central event queue in a *pipe-task* and an *LP* can execute as many events as possible before the next *LP* is scheduled. The algorithm follows the flow of events through the pipeline. This approach facilitates a very good cache locality. On one hand, when executing many events on the same *LP* there is a high probability that the destination *LP* (same destination all the time) state is in the cache of the processor that executes the task. On the other hand, when following the flow of events throughout the pipeline the probability of the event buffer being in the cache of the same processor is also higher.

To summarise, in a system using the CCT algorithm, scheduling is performed on three levels:

- ***Event scheduling***: the scheduling of events by the destination LP. In each session an *LP* attempts to process as many of the events that have been sent to it as possible.

- ***LP scheduling***: the scheduling of *LP* within a task. Depending on the type of task, the scheduling algorithm is based on a shared centralised future event list local to a *task* or it follows the event flow along the pipeline.

- ***Task scheduling***: the scheduling of *tasks*. *Task* scheduling is performed using a centralised shared task list where all the *tasks* ready for execution are stored. When available, a processor takes the next *task* from the list and starts processing it. This mechanism realises automatic load balancing and because the granularity of the work is large it prevents problems with processors contention for accessing the queue.

Because of its conservative approach, CCT imposes two extra conditions in addition to those introduced by the general *LP* modeling view. These conditions are shared by most conservative PDES synchronisation algorithms (Fujimoto, 1990):

1. To avoid deadlock situations, loops with minimum timestamp increment equal to zero are prohibited.

2. To assure the correctness of the execution, the timestamps of messages passed along a *channel* must be monotonically non-decreasing.

Although good results with this mechanisms have been obtained (Unger et al., 1999), they are dependent on a system with sparse *channel* connectivity and high message densities. As Xiao et al. (1999) noticed: "the goal of a general PDES simulation algorithm for low granularity events remains unreached". More information about the algorithm and other issues related to it can be found in the same paper.

2.5 Summary

This chapter has briefly surveyed the discrete event simulation field. Simulation is used in a variety of domains because it usually offers a convenient alternative for studying the time based behaviour of a complex system. DES assumes that changes in the system occur at discrete, possible random instances of time. Although it is usually simpler to implement, the increasing complexity of the application under investigation has forced researchers to look into possibilities for improving the performance of the simulation such as parallel execution on multiprocessor computers. Among all the approaches, PDES has gained in popularity because of its potential to exploit the parallelism in a model. There are two main mechanisms for this type of simulation: the conservative mechanism and the optimistic approach. Each of these has a number of strengths and weaknesses, and therefore there are numerous variations that aim at optimising different aspects of the classical algorithms. Many more or less successful implementations of these algorithms have been reported (Fujimoto, 1993). CCT is a new proposed algorithm based on the conservative scheme. Preliminary tests show encouraging results with CCT for particular types of problems.

CHAPTER 3: OBJECT-ORIENTED FRAMEWORKS

This chapter surveys the main concepts in the field of object oriented frameworks. Reuse of software has been a goal of software engineering for decades. With the emergence of object-oriented (OO) techniques, important progress towards achieving this goal was made possible. An object-oriented framework is essentially the design of a set of objects that collaborate to carry out a set of responsibilities in a particular problem domain. OOFs attracted attention from many researchers and software engineers for their ability to facilitate the reuse of larger components. Although OOFs have been successfully used for a variety of domains, problems and impediments associated with frameworks still exists. Being aware of and understanding these issues can help a designer to minimise their weaknesses and fully exploit their strengths.

The main goal of this chapter is to facilitate the development and analysis of OOF by presenting their main underlying concepts along with the main advantages and disadvantages associated with their use. It starts in section 3.1 by identifying the need for software reuse and the role OOFs in this context. Section 3.2 briefly introduces the OO paradigm, as frameworks rely heavily on its mechanisms such as inheritance, object composition, polymorphism. In section 3.3 OOFs are introduced with their definition, classifications and internal components as presented in the literature. The iterative process of building a framework with its main phases is then described. The next two sections point out some consequences of using frameworks, highlighting some of their strengths and weaknesses. The last section (section 3.4) briefly discusses the applicability of OOF to the DES problem domain.

3.1 Software Reuse

One of the main challenges contemporary organizations increasingly face is generated by the existing gap between the complexity of software development and the state-of-the-art of software engineering tools. While computing power and network bandwidth have increased dramatically over the past decade, the design and implementation of complex software remain expensive and error-prone (Fayad and Schmidt, 1997). Despite some gains, the software industry still faces long development cycles, which often result in software products that do not address business problems adequately. One of the main causes is that much of the cost and effort are generated by the continuous rediscovery and reinvention of core concepts and components. Software reuse has therefore become a goal in software engineering for decades.

Reusing software is not a simple assignment (Johnson and Foote, 1988),(Opdyke, 1992). Important progress towards achieving this goal was made during the 1980s with the emergence of the object-oriented (OO) paradigm. OO has the potential of increasing the reuse of software components (objects) through inheritance and object composition mechanisms. As a result, the software industry has moved towards embracing object-oriented technology because of its potential to significantly increase developer productivity through reuse. Still, all the OO techniques provide reuse only at the level of individual, often small components. The more complex problem of reuse at the level of large components that can be adapted for individual applications was not addressed by the OO paradigm itself.

Frameworks, large OO structures that can be tailored for specific applications, carry the OO paradigm further by providing infrastructure and flexibility for deploying OO technology and enabling reuse at a larger granularity (Bosch, Molin, Mattson and Bengtson, 1997),(Taligent, 1996). Since their inception in late 80's, frameworks have attracted attention from many researchers and software engineers proving to be a promising technology that enables reuse of large scale components. Frameworks have been defined for a large variety of domains such as: multimedia (Posnak, Lavender and Vin, 1997), operating systems within computer science and financial systems (Bäumer,

Gryczan, Knoll, Lilienthal, Riehle and Zullighoven, 1997), process control systems within particular application domains (Doscher and Hodges, 1997), (Brugali, Menga and Aarsten, 1997), and many others. As more experience has been gained in developing OO frameworks, several research projects have been conducted to study their impact in the software development effort and to identify their problems and difficulties.

3.2 Object-Oriented Programming

The main obstacles encountered in traditional procedural programming languages such as: difficulty in extending and specializing functionality, difficulty in factoring out common functionality for reuse, barrier to interoperability, maintenance overhead (Taligent, 1996), have forced the software community to look for new approaches to software programming.

The object-oriented paradigm presents new techniques to facing the challenge of building large-scale programs. It originates with Simula, which was initially dedicated to solving simulation (model building) problems. Since then, OO technology has been exploited in a wide range of applications including databases, operating systems, distributed computing and user interfaces. The main benefits that confer such a broad range of applicability to the object-oriented approach as presented in Abadi and Cardelli (1996) are:

- ***The analogy between software models and physical models.*** The analogy with a physical system model has proved to be useful in the process of developing a software model. It makes the analysis of the problem more efficient.
- ***The resilience of the software models.*** Unlike the approach advocated by procedural languages, which emphasizes the use of algorithms and procedures, the design of OO systems emphasizes the binding of data structures with the methods to operate on the data. The idea is to design object classes that correspond to the essential features of a problem. Algorithms, factored in methods and encapsulated in objects, form natural data abstraction boundaries. The main consequence of encapsulation is that it helps focus on modeling the system structure rather than trying to fit a problem to the procedural approach of a computer language.

- ***The reusability of the components of the software model.*** Objects are naturally organized into hierarchies during analysis, design and implementation and this encourages the reuse of methods and data that are located higher in the hierarchy. Furthermore, this property generates all the other advantages associated with software reuse including low maintenance overhead, high productivity etc.

In spite of these advantages, designing a good OO software still remains difficult, one of the causes being the large variety of available design options and the trade-offs associated with them. Software design patterns, one of the hottest topics to emerge from the OO community, address this issue by offering a possibility for capturing and expressing design experience. Patterns capture knowledge about software construction that has been gained by many experts over many years.

Software patterns first became popular with the wide acceptance of the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1995). There are also a number of other publications that helped popularise patterns (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996), (Pree, 1995). Patterns have been used for many different domains, largely for software architecture and design.

One of the main advantages of the OO paradigm is that it promotes the reusability of software components. Researchers (Johnson and Foote, 1988), (Johnson and Russo, 1991), (Opdyke, 1992) have identified those attributes of object-oriented languages that promote reusable software:

- ***Data abstraction*** refers to the property of objects to encapsulate both state and behavior. The only way to interact with an object and to determine an object state is by its behavior. Thus, data abstraction encourages modular systems that are easy to understand.

- ***Inheritance*** is the sharing of attributes between a class and its subclasses (Abadi and Cardelli, 1996). It promotes code reuse, since code shared by several classes can be placed in their common superclass to be inherited and reused. The programmer can

define in this case a new class by choosing a closely related class as its superclass and describing the difference between the old and new classes. This style of programming is called *programming-by-difference* (Johnson and Foote, 1988). Another advantage of inheritance is that it provides a way to organize and classify classes, since sibling classes are usually related.

- ***Polymorphism*** is defined by Booch (1994) as:

  > "A concept in type-theory, according to which a name (such as variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways." (p. 517)

  In other words, a method can be invoked on an object without knowing the object's exact type. Because it works with a wider range of attributes, it is easier to reuse a polymorphic method than one that is not polymorphic. For example, the expression a+b will invoke different methods depending upon the class of the object in variable a. Operator "+" in this case is overriden in each class.

  Polymorphism allows an object to interact with other different objects as long as they have the same interface. It simplifies the definition of client objects, decouples objects from each other and allows them to vary their relationships to each other at run-time (Gamma et al., 1995).

Object-oriented languages have introduced a significant revolution in programming techniques. However, even if the programming job is made easier as the work is performed now at a higher level of abstraction with objects and class libraries, the programmer is still responsible for providing the structure and the flow control of the application. Therefore, reusability is achieved mainly at the class level and only rarely at a higher level (e.g. structural level). Object-Oriented Frameworks, as it will be seen in the next sections, carry the OO paradigm further by aiming at reusing software at a larger scale.

At the end it is worth mentioning that one of the main drawbacks of OO programming seems to be execution efficiency. OO specific operations such as creation and deletion of

dynamic objects, virtual methods, excessive usage of pointers, to name a few, can be very time consuming. Therefore, many system designers share the belief that traditional procedural programming should be used for better performance and OO programming for better quality (modularity, maintainability, decoupling, etc.). While this statement might be true in general there are also examples of OO programs that are faster than their procedural counterparts (Mostafari, 1998).

3.3 Object Oriented Frameworks

Most authors agree that OO Frameworks are reusable design for an application, or a subsystem, represented by a set of cooperative abstract and concrete classes. Yet, there is no generally accepted definition of a framework and its constituent parts. A widely accepted definition is given by Johnson and Foote (1988):

> "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes" (p. 23)

The key terms in the definition of OOF are:

- ***Set of classes***. The set of classes refers to a number of object oriented classes corresponding to the essential features of a problem domain.

- ***Design***. The design of an application defines the overall structure of an application, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control.

- ***Abstract design***. An abstract design is a design in which some of the components (classes) are abstract. Abstract classes define methods in terms of a few undefined methods that have to be implemented by the subclasses.

- ***Solutions for a family of related problems***. The solutions to a family of related problems usually have common elements. They can belong to particular business

units (such as data processing or cellular units) or application domains (such as user interfaces or real-time avionics).

- ***Reuse***. Software reuse is the use of existing assets in some form within the software product development process. More than just code, assets are products and by-products of the software development life cycle and include software components, test suites, designs and documentation.

- ***Granularity***. Granularity in this definition refers to the level of reuse. Low granularity reuse is the reuse of components while higher granularity reuse refers to design or analysis reuse.

- ***Usability***. Usability refers to the quality or state of being usable. There are many factors that contribute to the usability of a framework such as: adaptability, robustness, learning, performance, standardisation, help, etc.

A framework usually defines the overall structure of all the applications derived from it, their partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. The developer is only responsible for customizing the framework to a particular application. This consists mainly on extending the abstract classes provided with the framework.

Another important characteristic of frameworks is the inversion of control between the application and the framework on which it is based. The developer usually writes the code the framework calls (Johnson and Foote, 1988). This is a totally different way of thinking. In conventional systems, the developer's own program provides all of the structure and flow of execution and makes calls to function libraries as necessary. This contrast between the flow control in the conventional approach and in the framework approach is illustrated in Figure 3. Note that this is the main flow control. Framework and libraries can have both called and calling components alike.

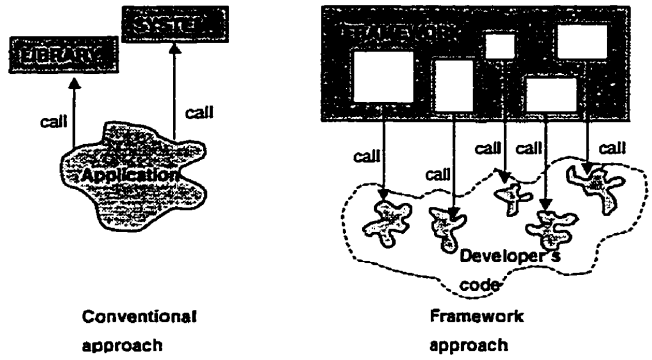


Figure 3 The flow control in conventional approach vs. framework approach

It is worth noting that there are tools that allow the OOF flow control to be implemented in procedural programming, so that frameworks do not necessary require an object-oriented programming language (Johnson and Russo, 1991). However, OO has the attributes that are particularly suitable for the framework approach.

3.3.1 Classification of Frameworks

There are many types of frameworks on the market, ranging from low level frameworks that provide basic system software services such as communication, printing, and file systems support, to very specialized high level frameworks for user interface or multimedia software components. Although the underlying principles are largely independent of the domains to which they are applied, a classification of frameworks by their scope is sometimes useful (Adair , 1995a),(Fayad and Schmidt, 1997):

- ***System infrastructure frameworks.*** Their primary use is to simplify the development of portable and efficient system infrastructure including operating systems, communication frameworks and frameworks for user interface. Being used internally within the organization, they are not typically sold to customers directly.

- ***Middleware integration or support frameworks.*** Their primary used is to integrate distributed applications and components. They enhance the ability of software to be modularized, reused and easily extended. Examples of middleware frameworks include message-oriented middleware and transactional databases.

- ***Enterprise application or domain frameworks.*** Their primary use is to support the development of end-user applications and products directly and therefore represent the base of enterprise business activities. They address different types of applications in a broad application domain such as telecommunications, avionics, manufacturing, education and financial engineering. In spite of the cost of developing and/or purchasing, enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly.

Another important classification is to consider the techniques used to extend a framework. From this perspective, frameworks range along a continuum between the two extremes:

- ***White-box or architecture-driving frameworks*** rely heavily on OO features such inheritance and dynamic binding. The framework is extended either by inheriting from framework base classes or by overriding pre-defined hook methods (Fayad and Schmidt, 1997). A white-box framework defines interfaces for components that can be plugged into it via object composition. However, the difficulty of using white-box frameworks resides in the fact that they require in-depth understanding of the classes to be extended. Another weakness, specific to subclassing in general, is the dependence among methods: e.g. overriding one operation might require overriding another and so on. Subclassing can lead in this case to an explosion of classes.

- ***Black-box or data-driven frameworks*** are structured using object composition and delegation rather than inheritance. They emphasize dynamic object relationships rather than static class relationships. New functionality can be added to a framework by composing existing (provided) objects in new ways to reflect the behavior of an application. The user in this case does not have to know the framework in-depth details, but only how to use existing objects and combine them. Black-box frameworks are therefore generally easier to use than white-box frameworks. On the other hand, black-box frameworks are more difficult to develop since their interfaces and hooks have to anticipate a wider range of potential use cases. Due to their predefined flexibility, black-box frameworks are more rigid in the domain they support. Heavy use of object composition can also make the designs harder to understand. Nevertheless,

many framework experts expect an increasing popularity of black-box frameworks, as developers become more familiar with techniques and patterns for factoring out common interfaces and components.

These two categories presented above are extreme cases because in practice a framework hardly ever is pure white-box or black-box, or has only called or calling components. In general, in a framework, inheritance is combined with object composition.

3.3.2 Framework Components

Conceptually, most of the authors (Demeyer, Meijler, Nierstrasz and Steyaert, 1997) consider that there are two main components of a framework:

- ***Framework contracts***. The common functionality in a specific domain is captured in the framework contracts. They formalize exactly which parts of the framework are to be reused. Thus, framework contracts impose a common structure for all the applications that use the same framework. The implementation and functionality of contracts are usually hidden from the user. However, because they form the skeleton of all applications, they have a direct impact on the performance and correctness of an application.

- ***Hot spots***. A variable aspect of an application domain is called a hot spot (Schmid, 1997). A framework is tailored for a specific application by implementing its hot spots according to the specific functionality of the application. Thus, different applications will differ from each other with regard to at least one hot spot. A hot spot allows a user to insert an application-specific class or subsystem. This can be done either by selecting the class or subsystem from a set of a set of predefined classes supplied with a black-box framework, or by extending the abstract classes as in a white-box framework case (Schmid, 1997) (Figure 4).

Hot spots provide the mechanism for extending a framework and therefore their design has a big impact in the usability of the framework in general, especially in its flexibility and variability. Due to their importance, hotspots are further analysed in the next section.

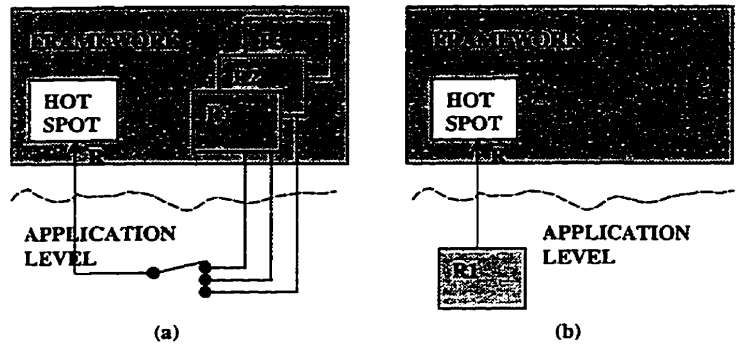


Figure 4 The hot-spot mechanism: (a) in a black-box the user has only to choose a class or a subsystem from the set supplied by the framework and (b) in a white-box system the user has to actually build the class or the subsystem to be used by the framework

The physical structure of a framework is composed as described by Bosh et *al.* (1997) of two main components:

- ***Core framework design*** comprises both abstract and concrete classes and describes the typical software architecture for a class of applications in the problem domain. Hot-spots are implemented by the abstract classes in the framework visible for the framework user. The user has to extend these subclasses or to use subclasses provided by the frameworks in the black-box framework case.
- ***Framework internal increments*** represent optional libraries that might accompany the framework. Their role is to make the framework more usable. Concrete subclasses used in black-box extensions or other classes that implement a specific common behavior are part of the internal increments of a framework.

3.3.2.1 Hot-Spots

Hot spots have a big impact in the reusability and flexibility of a framework and therefore it is worthwhile to explore them in more details. Schmid (1997) suggests that the variability required from a hot spot can be classified by the following characteristics:

- The common responsibility (R in Figure 4) that generalises the different alternatives
- The different alternatives that realize R

- The kind of variability required. This variability can be considered for example in alternatives with a common interface but different implementations, or alternatives with uniform service over different structures and so on.
- The multiplicity that gives the number and structuring of the alternatives that may be bound to a hot spot. It is directly related to the previous characteristic in the sense that usually the kind of variability dictates the number and structure of the alternatives.
- The binding time represents the point of time at which an alternative is selected. This time is either the time of creating an application or the run time. In the first case the application developer realizes the binding while in the second case it is the end user responsibility to do it either once or repeatedly.

Structurally, a hot spot is typically composed of a base class and a number of subclasses as illustrated in Figure 5:

- An abstract base class, which defines the interface for common responsibilities
- Concrete derived classes, which implement application specific alternatives
- Possibly additional classes and relationships

In the case of a white box framework, the application programmer has to implement the derived concrete classes. In contrast, a black box framework provides all these concrete classes and the user is responsible for choosing the appropriate ones and combining them to obtain the functionality required by the application.

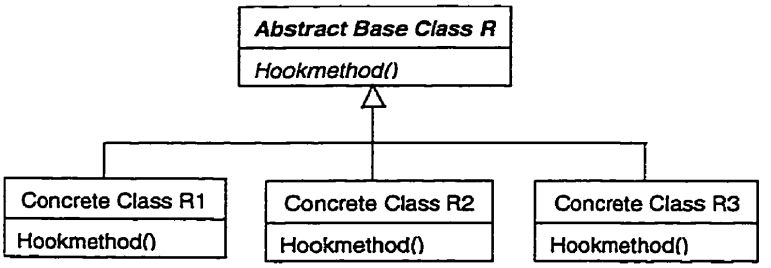


Figure 5 Example of hot spot subsystem (Schmid, 1997).

A hot spot usually contains or has attached a polymorphic reference typed with the base class. The user binds the hot spot by setting the reference to a subclass object of the base

class. This object can be from a prefabricated set supplied with the framework in black-box case or can be built by the application developer by extending the base class. Methods in the base class are specialized in a child class and every call to such a method will be dynamically bound, via its reference, to the subclass method executed. Therefore, a hot spot subsystem introduces variability that is usually transparent to the remainder of the framework.

3.3.3 Developing Frameworks

The task of designing a framework is difficult:

> "If applications are hard to design, and toolkits are harder, then frameworks are hardest of all." (Gamma et *al.* 1995, p. 27)

Part of the problem comes from the inherent conflict between reuse and tailorability. Packaging software components that can be reused in as many contexts as possible and designing software architectures that are easily adapted to target requirements are usually contradictory requirements. Another difficulty resides in understanding the problem domain and finding the appropriate abstractions (Johnson and Russo, 1991).

Building a framework is essentially an iterative process (Johnson and Russo, 1991), (Adair, 1995b) as frameworks more than any other software product are subject to change and refactoring (Opdyke, 1992). Bosch et *al.* (1997) identify three phases in forming the framework development process. They are illustrated in Figure 6 and are described as follows:

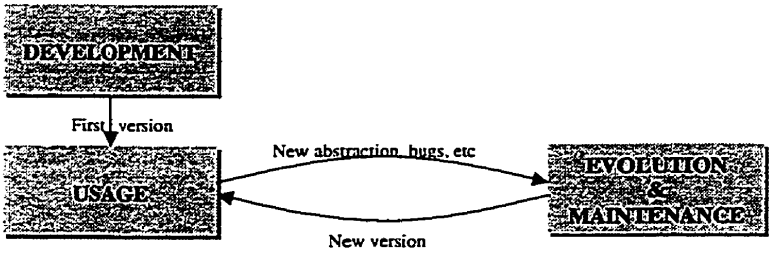


Figure 6 The process of developing a framework

1. ***The framework development phase.*** It is often the most effort consuming phase and its outcome is the initial version of the framework. Frameworks can be thought of as abstractions of possible solutions to a class of problems. Since people develop abstractions by generalizing from concrete examples, looking at existing solutions can identify opportunities for new frameworks. The process of identifying abstractions in a problem domain is different depending on the experience a designer has accumulated in that domain.

   One approach assumes that the developer has expertise in the problem domain and thus the analysis phase consists of identifying abstractions in families of applications that have been already built. After the abstractions have been identified, a first version of the framework can be developed. The existing applications are then built using the framework, and any necessary revisions to the framework, are considered.

   The second approach assumes that there is no previous experience in the problem domain. In this case, domain experts will not understand how to codify the abstractions that they have in their minds, and programmers will not understand the domain well enough to derive the abstraction. It is necessary therefore to first develop a set of applications in the problem domain. The more applications are built, the more apparent common abstraction will become.

2. ***The framework usage phase.*** It is sometimes referred to as the framework installation or application development phase. In this phase the framework is used in building applications.

3. ***The framework evolution & maintenance phase.*** It uses knowledge gained during the previous phase in order to change the current version of the framework. A framework evolves as:

   - errors are reported from shipped applications
   - new abstractions are identified due to changes in the problem domain or changes in the business domain
   - common elements become obvious in many applications.

It is important to note that the emphasis in this model is on the *process* of building frameworks. In contrast, Roberts and Johnson (1996) regard a framework as a *product* of such a process and propose a model for its evolution, as illustrated in Figure 7. The model consists of several patterns. As illustrated in Figure 7, at a particular moment in framework's evolution, one or more of these patterns components can be part of the framework.

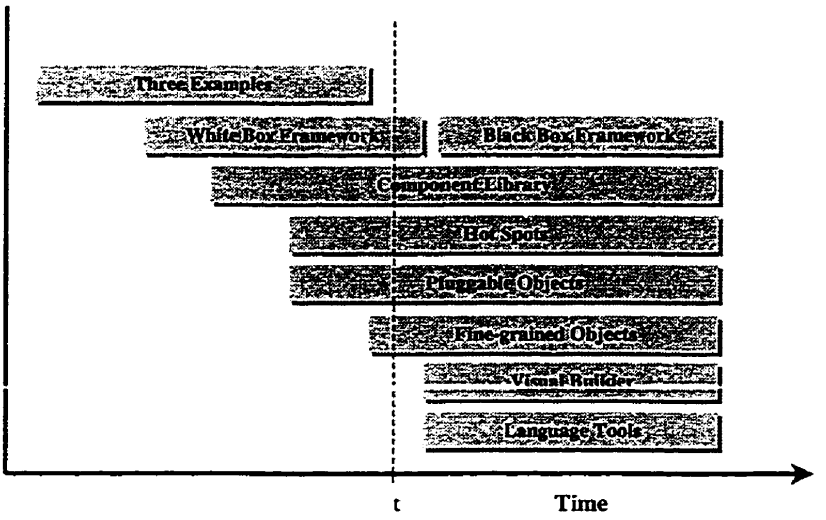


Figure 7 Typical framework evolution (Roberts and Johnson, 1996)

In this model, a framework usually -- but not necessarily -- follows the path illustrated in Figure 7. The patterns are described as follows:

Three Examples (Initial Development)

Roberts and Johnson (1996) propose building three examples in the problem domain in order to identify abstractions to be captured by the framework. It is one of the approaches for initial framework development used especially when there is no previous expertise in the problem domain. Three is suggested as a minimum number of examples to be considered but the more applications are used the more evident some of the abstractions in the problem domain become.

White Box Frameworks

As described in section 3.3.2, white-box frameworks use inheritance for tailoring a framework to a specific application. While developing subsequent applications, stable parts

that are not usually overriden can be factored out in abstract classes and included in the framework. They form a white box framework because the programmer has to subclass these abstract classes in order to use them.

Component Library

Common objects that are often used in many applications can be stored in a library for future reuse. A library usually contains concrete objects that can be directly used in an application. A framework with a good library of concrete components will be easier to use. Various applications develop concrete classes for tailoring a framework to a specific application. The component library of a framework is the result of accumulating such concrete classes that can be reused in future applications. While at the beginning every concrete component can be included in the library in the long run, only those that are often used remain.

Hot Spots

If the hot spots of a framework are scattered across an application, it is difficult to track them down and change. Therefore, by gathering the code that varies into a single location (ideally a single object) it will both simplify the reuse process and show users where the designers expect the framework to change.

Pluggable Objects

Pluggable objects refer to those subclasses that differ in trivial ways. The idea is that the new classes, no matter how trivial, increase the complexity of the system. Therefore, it is preferable to use adaptable subclasses that can be parameterized with messages to send, indexes to access, blocks to evaluate, or whatever else distinguishes one trivial subclass from another.

Fine-grained Objects

One way of reducing the number of possible concrete objects in the library (and implicitly the complexity of the system) is to break objects into finer and finer granularities. A composition of these objects can recreate the desired behavior of any original class. This will reduce the code duplication, as well as the need to create new subclasses for each new application.

Black-box Framework

As already mentioned, a black-box framework is a framework in which one can reuse components by plugging them in. The main advantage of using black-box frameworks is that the user can avoid programming and allow the compositions to vary at run time. On the other hand black box frameworks are harder to develop because they have to implement all the functionality required by different applications in the problem domain.

Visual Builder

With a black box framework, an application can be created entirely by connecting objects of existing classes. A graphical program that allows the user to specify the objects in an application and how they are connected seems to be the next logical step especially considering the fact that domain experts are rarely programmers. At this point they should be able to create applications by simply manipulating images on the screen.

Language tools

The visual domain-specific language that was just built requires, just like any other programming language, features to assist in the programming and debugging processes.

3.3.4 Advantages of Using Frameworks

The popularity of frameworks is justified by their strengths in enabling the reuse of software components, design and analysis. In the following, a number of advantages of using frameworks are described (Landin and Niklasson, 1995):

Analysis, design and code reuse. The main advantage of frameworks is that they capture the programming expertise necessary to solve a particular class of problems. Programmers use frameworks to obtain such problem-solving expertise without having to develop it independently. Frameworks allow reuse not only at the coding level but also, more important, at the design level and even at the analysis level (Johnson and Russo, 1991). It reuses analysis because it offers a guideline on breaking down a large problem into smaller problems and it describes the kinds of objects that are important. It reuses design because a framework introduces design constraints (component interfaces and communication algorithms) that any implementation must satisfy. Finally, it reuses code because the

implementation of a new component can inherit most of its behavior from an abstract class and can use components from the library that might accompany the framework.

Development time. Once a framework is understood, the development time for an application should drop because the code and the design can be reused. Besides, the modeling and the design are easier since frameworks offer guidance in developing the application.

Maintenance. This is another consequence of reusing the code in a framework. It is easier to maintain several applications built on top of a framework than the same applications built independently. This is a consequence of the fact that a framework imposes a similar design structure and functionality on all its applications in contrast with the design of the independent applications.

Testing. Testing an application that uses a framework is limited to the new modules introduced by the application extensions assuming that the framework works and is used correctly.

Reliability. As the framework is used, it is also tested more often and more bugs and errors are reported and solved. This makes applications built using stable frameworks more reliable compared to those developed from scratch.

Company standards. Ideally, a framework not only implements a proven design solution but also enables company standards. Standard constrains on the code and design, set by the framework have to be followed by its applications.

Frameworks embody expertise. As frameworks embody expertise, problems are solved once and the solution is used consistently. This enables software developers to concentrate on their particular problem domain and rely on the framework to provide consistent services.

Improved consistency and compatibility. Applications using a framework are more likely to be compatible with each other.

3.3.5 Drawbacks of Using Frameworks

OOFs do not claim to be the "magic formula" that will provide all the above advantages at no cost. There are a number of challenges that must be addressed in order to employ frameworks effectively (Fayad and Schimdt, 1997):

Framework development effort. As mentioned in previous sections, developing reusable frameworks for complex application domains is a very difficult task. Very often only expert developers posses the skills required to produce frameworks successfully.

Learning curve. Learning to use an OO framework typically requires considerable effort. Depending on the complexity of the framework, it might take several months to become highly productive with a specific framework. Sometimes, hands-on mentoring and training courses are required for improving the learning process. However, this effort might not be cost-effective, unless the effort required for learning a framework can be amortized over many projects. There has been extensive work on improving the learning curve. Proposed solutions range from methods and rules for improving the documentation (Johnson, 1992), (Adair, 1995b) to delivering frameworks with concrete examples that can be used in the learning process (Gangopadhyay and Mitra, 1995).

Integrability. As applications become more and more complex, they will be increasingly based on the integration of multiple frameworks, class libraries and existing components. Since many earlier generation frameworks were designed only for a specific problem domain, difficulties may be encountered in integration with other frameworks specialised for other domains.

Maintenance. As discussed in the previous section, applications built with frameworks are easier to maintain because assuming that the framework is correct, only the extensions that the applications introduce have to be maintained. However, the other side of the coin is that frameworks evolve over time and thus applications must evolve with them. This introduces an overhead in the application maintenance. It is desirable therefore that a framework is not extensively used until it is proven itself (Johnson and Russo, 1991). It is

probably better to first use the framework for some small pilot projects as test cases for the framework developers.

Validation and defect removal. Similar to maintenance, testing has also a side effect. Although a well-designed modular framework can localize the impact of software defects, validation and debugging can be a difficult task. This happens because bugs are introduced into the framework from many possible sources such as failure to understand the requirements, overly coupled design, or an incorrect implementation. Since these sources of errors are common for any software development effort, it is difficult to distinguish bugs in the framework from bugs in the application code.

Efficiency. Mechanisms that frameworks rely on, such as dynamic binding, employ additional levels of indirection. The use of dynamic binding improves the generality and flexibility of the framework but reduces efficiency. This also applies to other OO specific operations such as dynamic creation and deletion of objects, virtual methods etc.,

Standards. Currently the lack of accepted standards for designing, implementing, documenting, and adapting frameworks impedes them from being truly effective across multiple application domains. A framework developed in one language cannot be used for applications that use other languages. It is important for companies and developers to work with standards organizations and middleware vendors to ensure that the emerging specifications support true interoperability and define features that meet their software needs.

3.4 OOF Applied to the DES Problem Domain

Simulation products on the market come in two main flavours: simulation languages and application-oriented packages. Simulation languages offer more flexibility than the simulation packages (Maria, 1997). However, their effective use requires programming expertise.

The majority of DES packages adopt a framework structure. The DES world is very suitable for the framework approach because the basic mechanisms such as sending and

receiving events are very similar in all DES applications and can be captured by frameworks. There are many examples of white-box DES frameworks that implement different modeling views: such as GTW implementing the *LP* view (Das, Fujimoto, Panesar, Allison and Hybinette, 1994), TeD for process driven (Perumalla, Oglieski and Fujimoto, 1996)(Perumalla, 1996), JSIM for event-driven and process-driven (Miller, Nair, Zhang and Zhao, 1997) etc. In contrast, most of the commercial simulators such as OPNET (1998) or MedModel (1998) are black-box frameworks that address specific problem domains (telecommunication networks or health care simulations). Another example of black-box framework for the simulation of ATM networks is the ATM-TN simulator (Unger, Covington, Gburzynski, Gomes, Ono-Tesfaya, Ramaswamy, Williamson and Xiao, 1995).

TeD is an interesting example of a simulation framework, designed for computer network simulations. The two sets of orthogonal concepts that are distinguished in TeD are the framework itself (MetaTeD) and the hotspots implemented by the user (external language). MetaTeD implements the contracts (body) of the framework that consists in defining the various network elements, how they interact to each other etc. All the complexity of the underlying parallel synchronisation mechanism is captured in MetaTeD and hidden from the user. An interesting particularity of the TeD framework is the idea of using different programming languages for implementing its hotspots. Although MetaTeD is written in the C++ programming language, it can be extended (combined) with any regular general-purpose programming language.

As a consequence of simulation packages following the framework approach, they share some of OOF general advantages such as: easier to learn compared to simulation languages, modeling constructs closely related to the application, natural framework for simulation modeling and conceptualization to name a few. Another advantage is that some of the complexity of the implementation, whether for sequential or parallel execution, is hidden from the user. There is therefore a preference for these simulation frameworks over simulation languages.

3.5 Summary

This chapter has presented a brief overview of the state-of-the-art in the field of object-oriented frameworks. A number of issues concerning OO frameworks have been surveyed, with the emphasis on understanding frameworks composition, their strengths and weaknesses.

Frameworks do not require object-oriented programming. However, developing a framework using such a language is preferable since the OO paradigm has attributes such as: data abstraction, inheritance and polymorphism that support reuse of software elements. OOFs carry the OO paradigm further by providing infrastructure and flexibility for deploying OO technology.

Frameworks have been used extensively in a variety of problem domains. Several common components are found in all these types of frameworks. Framework contracts and hot spots are the most general components of a framework. The design of the contracts has a big impact of the correctness and efficiency of the framework's applications while the design of its hot spots influences its flexibility and extensibility.

A model of framework evolution was then presented along with some patterns that usually occur in the lifetime of a framework. Also, some of the strengths and weaknesses of frameworks were discussed in the last sections of the chapter. The main goal was to make the reader aware of the benefits of frameworks as well as some pitfalls that can impede the successful development and use of frameworks.

Finally, the last section of the chapter briefly discussed the applicability of OOF to DES problem domain. Examples of DES packages described in the previous chapter have been identified as following the OOF approach.

CHAPTER 4: SIMKIT - AN OOF FOR DES

Having laid out some of the underlying principles of DES and OOF, the main goal of this chapter is to introduce SimKit - the simulation package that we wish to extend. SimKit was developed at the University of Calgary and is aimed at fast parallel and sequential execution of DES applications. One of the key aspects of extended SimKit is that it has been identified as being an OOF within the DES problem domain. Based on this fact, possible directions in SimKit's evolution are identified and framework techniques and knowledge are applied for accomplishing this task.

Section 4.1 presents a detailed description of SimKit. This includes its initial requirements and the main components in the SimKit application programmer interface (API). Section 4.2 describes the main execution phases of a SimKit application. In the last part of the chapter (section 4.3) SimKit is identified as an OOF and it is analysed from this perspective. The analysis consists of identifying the two main OOF components in SimKit, contracts and hot spots, as well as some advantages and disadvantages derived from SimKit being an OOF. The discussion continues with the presentation of some of the factors that can effect SimKit's evolution and based on the experience with OOF, the "three examples" approach presented in the previous chapter is chosen for extending the framework.

4.1 SimKit

SimKit is a high performance discrete event simulation package developed at The University of Calgary (Gomes et al., 1995) as part of the TeleSim project (Williamson, Unger and Xiao, 1998). Its goal is:

> "to provide an event-driven *logical process* modeling interface that facilitates the effortless building of application models for sequential and parallel simulation with high performance execution capabilities" (Gomes et al., 1995, p. 707)

SimKit represents a common interface for several simulation executives. Figure 8 outlines the architecture of a typical application that uses it. SimKit is used directly by an application and it can be configured with a number of kernels or executives that run on several platforms. The sequential kernel based on a central event list, CelKit, performs well on a variety of UNIX and Personal Computer platforms. Two parallel versions can run using a conservative algorithm or an optimistic one. WaiKit, the conservative kernel is an optimised CMB kernel developed at the University of Waikato (Cleary and Tsai, 1996). WarpKit is the optimistic kernel based on Time Warp mechanism and it has been developed at The University of Calgary (Xiao and Unger, 1995). Both implementations support efficient parallel execution on shared memory multiprocessor hardware such as the Silicon Graphics Power Challenge and the Sun Sparc-Server 1000/2000.

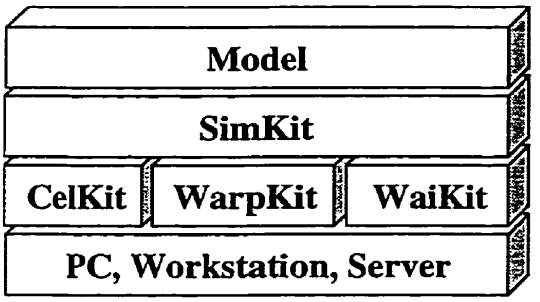


Figure 8 The architecture of an application that uses SimKit

The high level requirements that have driven SimKit's design and development and how the current implementation meets these requirements can be summarized as follows (Gomes et al., 1995):

1. ***Efficient event-oriented logical process view of simulation.*** SimKit is a high performance simulation package that adopts the *LP* modeling view of the simulation. Performance is the main driving force in SimKit and this is reflected in many of the design and implementation decisions. The *LP* view that SimKit implements is an

example in this sense. The *LP* view is very efficient, in spite of the fact that there is some debate whether it is the best approach for modeling a simulation problem (Page and Nance, 1994). There are two reasons why the *LP* view of simulation enables high performances. First the scheduling of events via invocation of the corresponding LP's event-processing member function is more efficient than the costly context-switching approach in a process-oriented model. Second, the memory requirements are lower as *LPs* are active only for the duration of an event i.e. there is no need for a per *LP* stack in the simulation. Besides the LP view, performance was also increased due to optimizations of the three engines. The FEL in the sequential simulation control system, for example, uses a splay tree implementation. The parallel versions contain variations of the conservative and optimistic mechanisms, optimized for execution on shared memory multiprocessors.

2. ***Transparency with respect to the underlying simulation kernel.*** This requirement refers to the compatibility among versions of the framework configured with different kernels. SimKit manages to hide the complexity of the engine, whether sequential or parallel, from the user presenting a common application programmer interface (API). Running a model using any of these kernels requires a small number of modifications in the application. Most of the required changes are due to the fact that a parallel version requires extra operations for partitioning and, in the optimistic case, state saving operations. This compatibility among different kernels is important because as mentioned in chapter 2, none of the PDES schemes implemented by these kernels can provide good performance for all types of applications. Compatibility allows the underlying kernel to be easily changed when the performance of an application is not satisfactory. Evolving schemes and feature extensions can also be easily integrated into the simulation control engine without affecting the SimKit API.

3. ***Embedded in a general-purpose language that supports a wide range of applications.*** The SimKit package is implemented in C++, a commonly used general-purpose object-oriented programming language. The applications that use SimKit are also required to be written in C++. The advantage of using a well-known language is that the

programmer is not forced to learn a new language and can instead concentrate on the system modeling work.

4. ***Ease of use***. SimKit, because of its simplicity is relatively easy to learn. Problems that might arise are because of the constraints the *LP* view imposes on the developer such those regarding the input-output handling during execution, error handling, prohibition against using global variables, etc. However, a large number of different simulation applications from a variety of areas have proven that in spite of the restrictions and constraints it introduces, the *LP* view of modeling can still be successfully used.

4.1.1 SimKit API

The SimKit API consists of three classes, one type and several convenience functions and macros. The three classes are `sk_simulation` for simulation control, `sk_lp` for modeling application sub-space behavior and state transitions and `sk_event` for modeling the interaction between *LPs*. These classes are directly derived and used in an application to implement a desired behaviour.

4.1.1.1 The sk_simulation Class

The `sk_simulation class` provides the initialization interface to the SimKit kernel (Figure 9). The user seldom, if ever, needs to derive classes from this class. Only one object of this class is allowed to exist at one time per program. The `sk_simulation` class offers different methods for returning the simulation current and end time, the size of largest event in the system, the different trace and debug flags. The `initialize()` method is automatically called by the kernel at the beginning of a simulation. Its role is to set the simulation parameters from the command line or from a configuration file such as the name of the experiment, simulation end time, debug, and trace flags of the simulation. There are also a number of methods used for setting these parameters. The main method that starts and runs the simulation is `start_simulation`. The `n_pes()` method returns the number of processors in the system. It has no implementation with CelKit kernel.

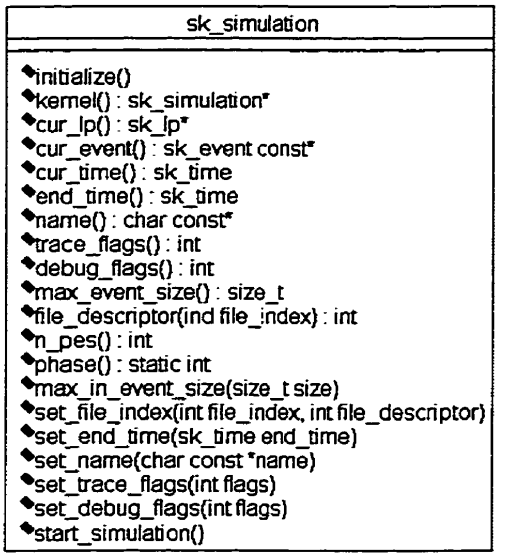


Figure 9 `sk_simulation` class methods

4.1.1.2 The sk_lp Class

The `sk_lp` class implements *LPs* in SimKit. In order to construct a simulation model the user implements his/her own *LP* classes derived from `sk_lp`. Each child of the `sk_lp` class represents one particular type of *LP* in the system. There might be more instances of a `sk_lp` subclass in the system, but all the *LP* objects can only be created at the beginning of the simulation.

Two of the methods (`alloc` and `dealloc`) in this class deal with memory allocation. They form the only allocation mechanism that is guaranteed to work correctly during parallel simulation execution.

For the user, the most important methods in this class are those that he/she has to implement. The three virtual methods are (Figure 10):

- `process(sk_event const *ev)` method is the place where the user specifies an LP's behaviour, that is, the activities that are performed by the *LP* when receiving an event. The kernel delivers an event to an *LP* by invoking this method with the event object as parameter. One approach is to implement this method as a large case statement with the event type specifying the branching of control. In this way it can mimic the behavior of the *LP* for different types of events. Another alternative is to

define a larger number of different LPs in the model each handling only few different events.

- `initialize()` method is called by the kernel once for each *LP* object at the beginning of the simulation. Optionally this method can be used to set the LP's initial state and to schedule the first event(s), sometimes called seed events.

- `terminate()` method is called after the simulation is terminated and before the *LP* object is destroyed. It allows the user to wrap up the *LP* work collecting eventual statistics and reports.

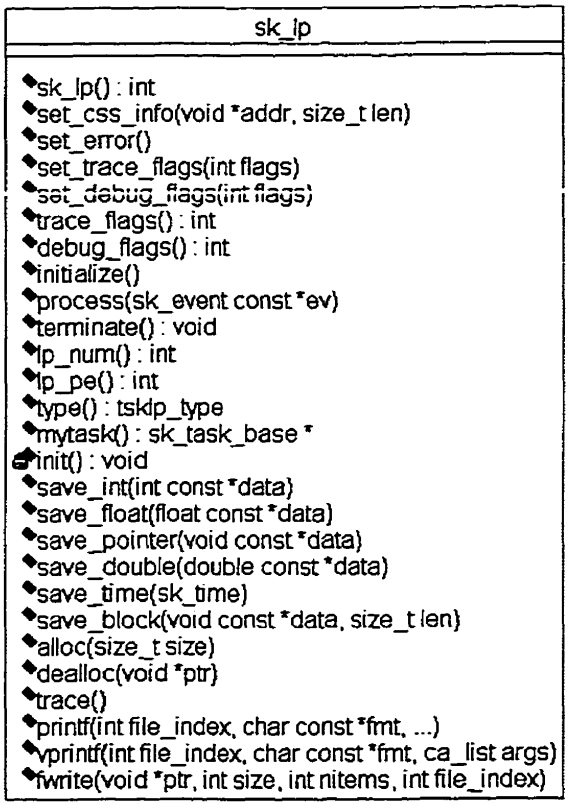


Figure 10 The visible structure of `sk_lp` class

Other methods set or read debug and trace flags, transparently deal with memory and input/output operations, both sequentially and in parallel, or save the state of a variable when used with the WarpKit kernel (Figure 10).

4.1.1.3 The sk_event Class

The `sk_event class` is the base class for all the event classes in an application. An *LP* (source *LP*) communicates with another *LP* (destination *LP*) by creating an event and sending it to the destination *LP* to be processed at a specific time. The user derives from this class in order to include application-defined content of event. It is not necessary that events correspond to real events in the system. An event, for example, may simply be a query message to determine the value of another LP's state as no shared variables are allowed.

There are just a few methods in this class as little functionality is associated with an event. For performance reasons, the `new` operator in `sk_event` class overrides the C++ `new` operator. All events that are to be sent must be allocated using this operator. The most important method in this class is `send_and_delete` that sends the current event to the specified *LP* to be received at the specified time. Once an event is sent, it is effectively deleted from the sender's address space. The `send_and_delete` is the only event synchronization primitive that is necessary in the event driven *LP* view.

The other few methods refer to debugging (`trace`) or are access methods (`recv_time`) (Figure 11).

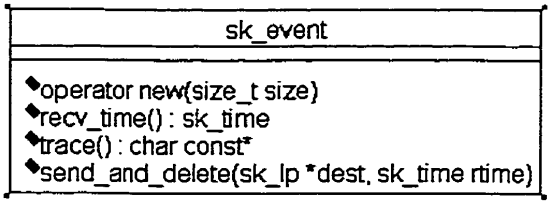


Figure 11 The structure of the public interface of `sk_event` class

4.2 Execution Phases

The execution of a SimKit application is divided into six phases (SimKit, 1995). It starts with a single thread of control executing on a single processor. Especially due to the parallel execution, there are a number of constraints and rules for each of the phases as listed in Table 2:

Table 2 SimKit execution phases

Ph.	Name	Description
1	Program Initialization	This phase occurs before the `initialization` method in `sk_simulation` class is executed. Constructors of static objects such as the simulation event-list are called during this phase. The function main is executed during this phase, the application model command line arguments are stripped off and the `sk_simulation` object is constructed, either statically or dynamically. No events may be created or sent during this phase. No *LPs* may be created during this phase.
2	SimKit and Model Global Initialization	This phase starts with the execution of the `initialize` method of class `sk_simulation`. This method strips and uses some of the command line arguments. The *LPs* that compose the model are instantiated using the `new` operator and allocated to processors. Allocation of *LPs* to processors is static and may be optionally specified by the modeler via the LP's constructor. Any global, application data structures are usually built during this phase. No events may be created or sent during this phase. Phase 2 ends with passing control to the simulation run time system.
3	*Logical process* Initialization	After the program has instantiated all *LPs* and completed other global initialization, it calls the `initialize` method in each *LP* class exactly once. Simulation time does not advance during this phase and no events are received. The restrictions on the programmer are: (1) no *LPs* may be created during this phase, and (2) events must be allocated via the `new` operator in `sk_event` class. Other than these, no restrictions are imposed on the programmer (although care should be taken when operating on global information in parallel execution since starting with this phase, each *LP* is executed in parallel). This phase is usually used to send out seed events to start the simulation. These seed events are not received by their destination *LPs* until all *LPs* have been initialized and phase 4 has begun even if they are scheduled to be received at time 0.

4	Simulation Execution	SimKit kernel, whether sequential or parallel, begins dispatching events to their destination *LPs*. When an event occurs at an LP, the `process` method in that *LP* is called and its LVT advances to the time the event was scheduled to occur. The phase ends when one of the following conditions is met: ❑ there are no more events to process (normal termination) ❑ the simulation end time has been reached (normal termination) ❑ a user reported error occurs (abnormal termination) ❑ a SimKit error occurs (abnormal termination)
5	*Logical process* Termination	This phase starts when phase 4 ends normally. Each LP's `terminate` method is called. Simulation time does not advance during this phase and no events are received. The programmer is restricted from sending events and creating *LPs*. This phase is usually used to perform *LP* specific termination operations, such as reporting *LP* specific statistics.
6	Simulation Clean-up	When phase 5 ends normally, the main function of `sk_simulation` class returns. There are no restrictions on the programmer. This phase is often used to tally statistics and output final reports.

4.3 Extending SimKit

SimKit has been used for different DES applications since its initial release in 1995. A number of changes to remove defects, kernel optimisations, new features etc, have been made in different releases during this time. The following sections and the next chapter describe a process for extending SimKit using an OOF approach and a number of issues related to the design and implementation of a new release that incorporates these extensions. The purpose of these changes is to support a particular class of DES applications: *network model* applications as defined in chapter 1.

4.3.1 SimKit an OOF for DES

The presentation made in the first part of this chapter suggests that SimKit is an OOF type of software. In supporting this hypothesis, Table 3 sumarizes how the key terms in the OOF definition presented in Section 3.3 are addressed in SimKit.

Table 3 OOF key terms and SimKit

Key term	SimKit package
Set of classes	SimKit package implement a number of classes corresponding to the essential features of the DES problem domain. The main features of a DES application (the simulation experiment, LPs and events in the system, the FEL, etc.) are each modeled by classes in the SimKit package.
Design	SimKit defines the overall structure of an application. Thus, a SimKit application consists of a number of LPs that collaborate to each other by exchanging events during the experiment.
Abstract design	SimKit's design is an abstract one since `sk_lp` class is abstract. It contains the hook method `process`, which is a pure virtual method that has to be implemented by the user.
Solutions for a family of related problems	SimKit offers solutions for any DES and PDES problem.
Reuse & Reuse Granularity	SimKit promotes reuse at different granularity levels: • at the very low level of class reuse by reusing the components implemented by the classes in the API • at the design level by imposing design constraints that each application has to follow such that regarding shared attributes • at the analysis level by that a problem has to be decomposed in disjoint states modeled by LPs that can exchange messages etc.
Usability	A number of factors such as: robustness, performance, simplicity etc. make SimKit a usable package. However, one of the few factors that reduce this usability is the adaptability of the framework; SimKit requires a relatively high investment in order to be customized for a specific application.

SimKit is a white-box OOF since its applications use mainly inheritance and overriding. Two framework specific components can also be identified in SimKit:

- ***contracts*** are represented by the three engines: CelKit, WaiKit and WarpKit. Each of these kernels implements the same classical *LP* view. The main difference is that two of these engines, WaiKit and WarpKit, allow building and running PDES applications. If execution time is not a critical issue, SimKit can use the CelKit kernel and execute sequentially, otherwise it can be configured with one of the other parallel contracts. As mentioned in chapter 2, due to their different synchronisation algorithms, the WaiKit engine offers good potential for certain classes of problems while WarpKit is more suitable for others.

- ***hot spots*** in SimKit are represented by the three main classes that are used to define an application, namely `sk_simulation`, `sk_event` and `sk_lp`. The `sk_simulation` class is usually used directly and only occasionally the user would subclass it. The `sk_event` class can also be utilised directly as a simple event object but it is usually derived and used as a carrier for message information. The `sk_lp` class is the only hot spot in SimKit that must be derived in order to be used. Its `process` method must be overridden, otherwise the simulation application cannot work (events have to be sent out and processed). In order to insure that the user overrides the `process` method, it is implemented as a C++ pure virtual method.

  A very important observation here is that these hot spots are common to all contracts. There are only a few methods that have a corresponding implementation for only one of the kernels (methods for state savings for example). As a result few changes are required in the application when switching from one kernel to another.

The observation that SimKit is an OOF is very important because the experience with other types of frameworks can be applied to guide SimKit's evolution and maintenance. Another consequence is that SimKit shares the same strengths and weaknesses with all other OOFs. The following are a number of advantages and disadvantages associated with the SimKit OOF.

Advantages that SimKit can and does exploit:

- ***Analysis, design and code reuse***. SimKit reuses analysis because if offers guidance for analysing a problem based on *LP* view modeling. According to this model, disjoint PPs have to be identified in the physical system first, then their patterns of communication established, and the types of events that are exchanged in the system. These guidelines can significantly simplify the analysis of the problem. SimKit also reuses the design by putting constraints in the design of component interfaces and communication algorithms. Finally, it promotes code reuse since the classes it provides are directly or indirectly (through inheritance) utilised in an application.

- ***Development time***. Improvement in the development time is mainly a consequence of reusing. Therefore, the more built-in components provided with the framework the shorter the development time and effort.

- ***Testing and reliability***. These two benefits of OOF are related. They are a consequence of the fact that as SimKit is used more bugs and errors are found and fixed.

- ***Framework embodies expertise***. This is a main advantage of using SimKit. Different kernels that are used with SimKit are the result of intensive research work performed by parallel simulation specialists and programmers with less expertise in PDES field can use these solutions. Therefore an application programmer is able to implement a PDES application following SimKit rules and guidelines with limited parallel software development expertise.

Disadvantages that SimKit should and does try to minimise:

- ***Framework development effort***. Developing SimKit and especially its internal parallel synchronisation mechanisms required a sustained effort. Similarly with other OOFs, this work has been performed mainly by expert developers.

- ***Learning curve***. Although this is a weakness for many OOFs, it does not have a big impact in SimKit. Due to its simplicity, one can relatively easily learn to use SimKit

especially if he/she is familiar with the *LP* modeling view. This is the reason this aspect is in fact presented as an advantage of SimKit compared with other simulation frameworks.

- ***Maintenance***. This refers to the difficulties generated by changes in the framework as it evolves over time. This also is not a serious issue in SimKit since most of the changes occur internally in the contracts (kernels). One of the priorities was that changes in the hot spots of the framework be kept at a minimum level. As an example, although WaiKit requires information regarding the topology of an application, in order to maintain a common, stable, API for all the kernels, the initial SimKit API has not been changed to allow capturing this extra information. Instead, the WaiKit kernel reads this information from an external file for each application. The downside is that the topology in the external file has to be consistent with the one in the application and this can be error prone.

- ***Validation and defect removal***. Debugging and bug-removal in an application built on top of a framework are more difficult as explained in the previous chapter. In order to minimize this weakness, SimKit has different execution modes. When running in debug mode it checks the code for correctness more often and signals potential misuse. It also offers the possibility of running an application in the trace mode. This facility, although not completely implemented in SimKit, supports this technique of marking particular events so that their effect as they travel in the system can be observed and analysed.

- ***Efficiency***. Since efficiency is one of the main requirements for SimKit and also the primary goal of PDES, a great deal of work has been done in improving the efficiency. Whenever possible, OO related time consuming operations have been restricted or optimised.

4.3.2 Directions in Extending SimKit

As a PDES framework, performance is the driving force in SimKit's development and evolution. This is reflected in the continuous effort for optimising the existing kernels (abstractions) and implementing new, more efficient ones such as the new TasKit kernel. The second goal is to facilitate the development process of the applications that use the SimKit framework. Libraries that implement common elements in many applications fall in this category.

A framework evolution is influenced by several factors as mentioned in Section 3.3.3. In the SimKit's case, changes caused by reported errors were the subject of a numbers of previous upgrades. The main focus in this thesis is on extending SimKit as a result of the other two types of factors mentioned above: new abstractions due to changes in the problem domain and common elements in many applications.

4.3.2.1 New SimKit Abstractions Due to Changes in the Problem Domain.

SimKit was initially aimed at general PDES problems. This problem domain is vast and includes applications from a variety of fields. However, within the TeleSim group, SimKit was mainly used for telecommunication and network applications. As a result, narrowing down the problem domain to the *network models* domain enabled new abstractions that provide a more efficient execution of an application to be identified.

The main characteristic of a *network model* application as defined in the first chapter is the fixed connectivity defined by its static interaction graph. This means that every *LP* in the model has to know the other *LPs* it is going to communicate with, statically at the beginning of the experiment.

The TasKit kernel is the implementation of the CCT algorithm and it captures the abstractions in the new problem domain of *network models*. It was specifically designed for this type of problems and as a result, in most of the cases it outperforms the other available kernels. In Figure 12, the speedup of one of the initial versions of the TasKit kernel is compared with the speedup obtained with the existing parallel kernels for the

simulation of an ATM network scenario called NTN-3 (Unger et al., 1999). The NTN-3 network topology represents the Canada-wide ATM National Test Network (as of March 1996). It consists of 54 ATM switches and 355 traffic sources that generate different types of traffic. The corresponding model has 1381 LPs and generates $216*10^6$ events for the simulation of 5 seconds of real time. The speedups obtained when running the model on selected number of processors (1, 2, 4, 8, 12 and 16) is relative to the CelKit sequential kernel.

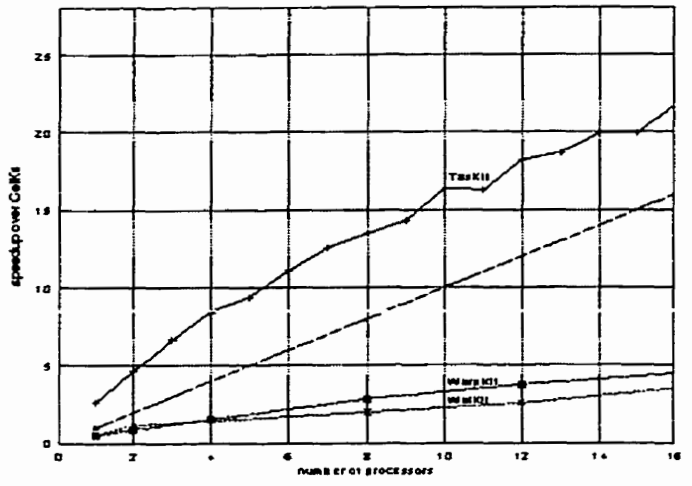


Figure 12 Performance results obtained with the TasKit, WaitKit and WarpKit kernels for the NTN-3 benchmark scenario (Unger et al., 1999).

This result is a strong argument in extending SimKit to support the TasKit kernel and some of the issues related to these extensions are discussed in the next chapter. New primitives in the SimKit API required by the TasKit are identified and their design is described.

The TasKit kernel is part of the core SimKit framework design. Two other changes in the core framework design are discussed in the next chapter. The `copy_and_delete` method optimises the process of dynamically creating and deleting event objects in a simulation experiment. The other new contracts in SimKit that proved to be useful for the network problem domain is an event cancellation mechanism. An implementation of this feature that has no effect on the performance of the framework is also presented.

4.3.2.2 New Common Elements in SimKit

SimKit is a white-box framework since it relies heavily on inheritance and dynamic binding. According to Roberts and Johnson's (Section 3.3.3) capturing the new common elements found in many applications in libraries that accompany the framework is usually the next step in framework evolution.

Both contracts and these libraries are part of the OOF. The main difference is that the contracts are part of the basic mechanism of the framework and their existence outside the kernel is not possible. In contrast, libraries that accompany the framework, sometimes also called framework internal increments, can be optionally utilised to make the framework more usable. Figure 13 illustrates the flow control among the core framework design, the framework internal increments and the code implemented by the user: the core framework calls the developer's code that might include calls to the libraries that accompany the framework.

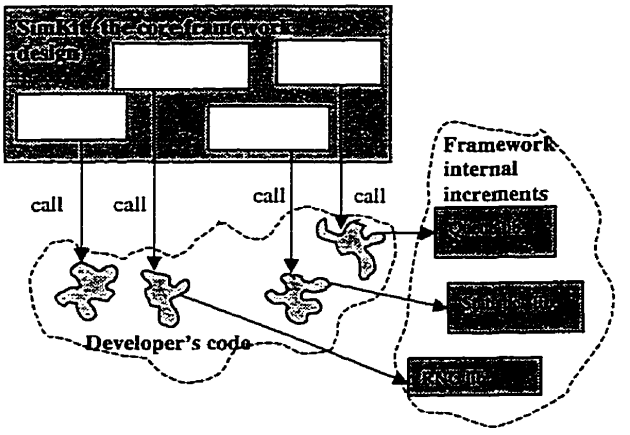


Figure 13 The flow control in the extended SimKit framework

Three types of libraries for the extended SimKit are considered: the queue library, the statistics library and the random number generator library. The components implemented (queues, statistics objects and random number generators) by these three libraries are very common in many simulation applications. These libraries are also incorporated in many other simulation frameworks. If the main goal of the modifications discussed in the previous section is improving the efficiency of the framework, the goal of implementing libraries is to make the framework more usable.

Another way of facilitating the development is to offer debugging features. Defect removal is one of the OOF weaknesses. In order to minimize its effects, the framework has to check the application for correctness. Another technique that facilitates debugging and defect removal is event tracing. Special events, marked as trace events, are followed in the experiment as well as the events and operations they generate. In some cases, event tracing can be a real useful tool in detecting errors in the model.

4.3.3 "Three Examples" Approach

An iterative prototyping design process has been chosen for SimKit framework. Iteration is inevitable as frameworks are supposed to be reusable. In order to test the reusability of the framework it has to be reused. Or, it is not possible to reuse a software program until it is written and working.

Due to the lack of detailed feedback from the SimKit's users, decisions regarding what exactly is needed in the extended version of the framework and how these changes should be designed were difficult to make. In this context, the "three examples" approach suggested by Roberts and Johnson (1996) was considered in the process of extending SimKit framework. The three example approach has a similar effect for OOF as case scenarios have for software with a user computer interaction interface. Choosing a number of different applications from the new problem domain and implementing them using the framework can reveal its weaknesses and suggest improvements. They also can suggest solutions for the interface and the design of the new desired feature in the framework.

Four *network model* applications have been chosen for investigating possible improvements in SimKit: the ATM-TN simulator (Unger et. al, 1995), a simple ATM model, an Ethernet protocol model and the Qnet model (Appendix A). These problems have been chosen because they represent common examples of problems very likely to require simulation studies and cover a large range of applications in the *network model* problem domain. Qnet for example is a very common synthetic benchmark for studying the performance of different PDES algorithms. While the ATM example and the ATM-TN

simulator are very suitable for the TasKit approach, the other two, the Qnet and Ethernet example, represent "the worst scenarios" for this kernel.

The four examples of application models (hereafter are referred to as the testing examples) were changed each time the SimKit API or the TasKit kernel was changed. They were used mainly to provide insights into the problem domain but also to guide and test the design and the implementation of the framework's new added features. Also, they can be useful as sample programs in documenting the framework.

At the end, note that none of the examples presented in Appendix A claims to offer the best solution for modeling the given problem. Their main role is only to help the designer gain more knowledge about the network application problem domain.

4.4 Summary

This chapter has introduced SimKit, the simulation package that has been used in the study presented in this thesis. SimKit is a DES simulation tool. Its purpose as well as its design structure places SimKit in the OOF category. Based on this fact, OOF techniques and knowledge were used for extending SimKit.

The detailed presentation of SimKit contained its initial high level requirements that have to be met by the extended version, as well as, a description of its API and the SimKit execution phases applications.

The focus of the discussion was then switched to extending SimKit and some of the issues related to it. First two main factors that cause OOFs' (and implicitly SimKit's) refactoring have been identified: new abstractions due to changes in the problem domain and common elements found in many applications. They imply changes in the framework contracts and hot spots, the latter ones being the main focus of the next chapter. The lack of detailed feedback from SimKit's users has led to choosing the "three example" for extending SimKit. This approach is used in building OOF for which there is very little information about the problem domain. It consists in using a number of examples that can suggest the new features and their implementation for the extended framework. They also are useful in debugging the new features of the framework and in documenting it.

CHAPTER 5: EXTENDING SIMKIT - DISCUSSION

This chapter briefly describes some of the main issues related to the design of an extended version of the SimKit simulation framework. Its goal is to discuss and analyze different design alternatives rather than to present a final version of the extended framework. The author believes that this approach is more valuable for two reasons. First, it aids better understanding of the underlying mechanism, the rationale behind choosing the final design alternative and the consequences associated with it. Second, this analysis can be used in solving other similar problems.

Analysing the design options for each of the new features implies discussing all their consequences. These consequences refer, in order of their importance, to the correctness of the algorithm, the performance, the usage and the simplicity of the API, etc. Although discussion does not include the details of the implementation (complexity or proprietary reasons), in order that these consequences are understood, a description of the underlying synchronisation algorithm or mechanism has to be provided.

The discussion is structured in two main topics according to the types of factors that caused modifications of the framework. The first topic pertains to changes in SimKit caused by new abstractions due to changes in the problem domain. It focuses mainly on issues related to supporting the TasKit engine (Section 5.1). This is also the most important extension of the framework. Other abstractions are presented in Section 5.2 and 5.3. The first one, `copy_and_delete` method aims at optimizing the event object allocation and de-allocation mechanisms. The second one proposes an event cancellation mechanism at the application level that can facilitate the implementation of a model. The solution that is

adopted has the advantage of introducing no penalties in performance with any of the four engines.

The second main topic consists of changes due to the implementation of common elements that became obvious in many applications. These components are implemented in libraries that allow their reuse. Libraries include libraries of queues, random number generators, statistics collection etc (Section 5.4). Finally, the last section (5.5) highlights the importance of a debugging mechanism in SimKit.

While the first category of changes has an impact mainly on the performance of the SimKit OOF, the second one is oriented towards reducing the development effort of the applications that use the framework.

5.1 Support for the TasKit Engine

The TasKit kernel is the implementation of the CCT algorithm outlined in chapter 2 and defined in detailed in (Xiao et. al, 1999) and (Unger et al., 1999). Preliminary results with TasKit demonstrated excellent performance for *network model* applications when compared to the existing sequential, CMB conservative or TimeWarp optimistic kernels. These encouraging results justify the effort to extend the SimKit framework in order to support the TasKit kernel.

TasKit's improved performance is in part due to a greater amount of information gathered from the application level. This information consists of the interaction graph of the application, the delays in the communication *channels*, the available lookahead at each *LP* etc. However, none of the previous SimKit kernels needs or uses this information[5]. Therefore, the SimKit API has to be extended in order to allow an application to pass this information to the TasKit kernel.

---

[5] In fact WaiKit kernel needs this information. However, as presented in the previous chapter, in order to keep the changes in the hot spots of the framework at a minimum level and have a common API with all the kernels, an alternative in which this information is gathered from an external, manually written file, was chosen.

The extended SimKit API becomes incompatible with the other previous kernels since some of its features and components have only been implemented in TasKit. As a result, the core of the following discussions is focused on how to support the necessary TasKit functionality with a small number of modifications in the SimKit API and without affecting TasKit's performance.

At the minimum, there are three new components in the extended SimKit: *channels*, *channel based LPs* and *tasks*. *Channels* are used to describe the interaction between *LPs* in the system. Events among *LPs* in the model can only be sent through *channels*. The `sk_node` class extends `sk_lp` class in the initial version of SimKit as the representation of *LPs* in a *network model*. It inherits the functionality of `sk_lp` class with the restriction that events can be sent only through *channels*. Finally, *tasks* have no correspondent in a model; they are used only for the sake of performance.

5.1.1 Channels

A *channel* is a unidirectional link between two *LPs*. All the events exchanged by *LPs* in the TasKit application are sent exclusively through *channels*. *Channels* are explicitly created by the application and map the arcs defining the *LP* interaction in the static graph of a *network model*.

Channels can store a number of events at any point in time. They implement a First-In-First-Out (FIFO) discipline and assume that the source *LP* sends events in a non-decreasing time stamp order. Each *channel* is characterised by its own simulation time and a delay. The simulation time of a *channel* is used to capture the current lookahead available in a channel. More specificly, it represents a lower bound on any future event to arrive on this channel. *Channel* delay on the other hand is a static attribute representing the minimum lookahead in a channel, i.e. the minimum difference between the receiving and the sending time of a message in the channel. The kernel uses the value of the delay in computing the clock of a channel.

Channels in TasKit are more complex than links in a CMB algorithm. The importance of *channels* for modeling network applications, as well as, the performance of the TasKit kernel are given by three key factors:

Channels define the LP interactions in the system. From the modeling perspective, channels map the arcs in the interaction graph corresponding to a network application. By using *channels*, accidental errors in which *LPs* mistakenly send events to other *LPs* they are not linked to can be reduced, assuming that *LPs* have been connected correctly. From the PDES point of view, *channels* allow the use of a class of conservative synchronisation algorithms including the CCT algorithm (as implemented in TasKit).

Channels, being unidirectional, define the causality relationships between LPs. In a simple situation like the one presented in Figure 14, the unidirectional *channel* between the two *LPs*, allow LP1 to send events to LP2. (LP1 is the source *LP* for the *channel* while LP2 is its destination LP). Because there is no *channel* to link LP2 to LP1, LP1 cannot get any feedback (any events) from LP2.

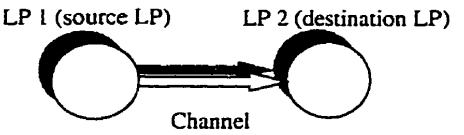


Figure 14 The causality relationship defined by an unidirectional channel

The clock in the *channel* represents the earliest possible time when a future event from LP1 can arrive at LP2. The value of the clock attribute, usually calculated by the kernel, depends on the LVT of LP1 and on the timestamp of the messages, if any, already in the *channel* (Xiao et al., 1999). In conservative execution, where no causality errors are allowed, this means that LP2 is limited to execute up to the clock time of the channel.

In order to obtain better cache performance and improve the efficiency of the execution, it is desirable to process a large number of events per session at each *LP* (Cleary and Tsai, 1996). Executing LP1 before LP2, is very likely to increase the value of the delay in the *channel* and therefore to maximize the time window for LP2. Ultimately, this allows a larger number of events to be processed at LP2.

In reality, a model is more complex that the one showed in Figure 14. Scheduling and executing *LPs* so that the number of events per session is maximized, has been the subject of intense research. However, for certain patterns of connectivity, the causality relationship introduced by *channels* allows the implementation of more efficient algorithms for scheduling *LPs*. This is the basic idea behind grouping *LPs* into *tasks* in TasKit, as seen later in this chapter.

Channels have a delay associated with them. A delay in a *channel* is the minimum lookahead between two *LPs*. Lookahead is a very important characteristic of an application that plays a critical role on the performance of conservative algorithms (Fujimoto, 1990). Also, TasKit as a member of the conservative algorithms family, prohibits any zero delay cycles (sum of the *channels* delay in a cycle is zero) in the model.

5.1.1.1 Status of a Channel

TasKit requires that all the possible interactions among the *LPs* in the system to be described by a static graph. However, it might happen that some *channels* are not used at all during a particular experiment. Imagine for example an application that models a big real ATM network. It is possible that during particular experiments with this network, some ATM sources do not generate cells. Their corresponding *channels*, although included in the static interaction graph, are not used at all.

An existing *channel* that is not used in an experiment might cause delays in the execution. Changing the interaction graph for each experiment by adding and deleting *channels* only to speed up the simulation is possible but it is not a viable solution.

The status of a *channel* is introduced to address this issue. Status is an attribute of a *channel* that reflects its behaviour during the experiment. It is statically set at the beginning of a simulation and cannot be changed afterwards. A *channel* can be in one of the two statuses:

- A *closed channel* although connecting two *LPs*, cannot be used during a simulation experiment for sending and receiving events. As a result, in order to speed up the

execution, the conservative synchronisation algorithm disregards a *closed* channel, as if it has not been created.

- An *opened channel* that connects a sender *LP* to a receiver *LP* allows events to be scheduled by the sender to the receiver.

The status of a *channel* is an alternative to changing the interaction graph for each particular experiment in order to optimize the execution. Instead of deleting particular *channels* in the application that are not used for a specific experiment, this mechanism only requires changing their status to *closed.*

5.1.1.2 Exploiting the Available Lookahead Using Channels

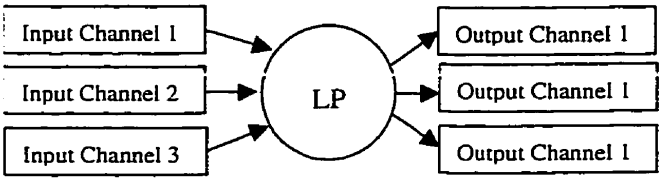


Figure 15 The *LP* configuration in TasKit

TasKit uses an optimised conservative synchronisation algorithm. In Figure 15 for example it is assumed that all the input and output *channels* of an *LP* are *opened.* Based on the clocks of the input *channels*, the algorithm maintains a local window time, which is an upper bound of the safe time that is found in an execution session. It repeatedly selects the non-empty input *channel* that contains the events with the lowest timestamp and process it until it reaches the time window limit. At this point the execution of the current *LP* is suspended. Before executing the next LP, the algorithm calculates the clocks for all its output *channels*. A detailed description of the algorithm and the proof for its correctness can be found in (Xiao et al., 1999). Needless to say that the larger the delay value in the output *channel* the larger the execution time window for next *LP* which means possible more events executed per session.

As shown above, the clock value of a *channel* captures the lookahead and it is calculated as a function of the *channel* delay, LVT of the source *LP* and the timestamps of the events in

the channel. However, there are cases when the lookahead suddenly increases during the simulation beyond the value computed by the kernel. Imagine for example that an *LP* that models an ATM source finishes generating all its ATM cells for an experiment. The available lookahead on its output *channel* becomes larger (equal to the simulation end time) but the kernel is not aware of it. In order to take advantage of this opportunity, the kernel has to be informed when such an opportunity arises.

Probably the simplest and most effect alternative is to advance the clock of the output *channel* to a specified value, anywhere up to the simulation end time. For this period of time, the *channel* although *opened,* becomes inactive and it cannot be used to send events through. By advancing the time, the source *LP* makes a promise that it will not send any event during that interval.

This mechanism is efficient because it can take advantage of the lookahead that becomes available in the application. Since the information regarding the lookahead comes from the *channels*' source *LPs*, logically only these *LPs* should be able to advance the clock in a channel.

The *channel* clock feature might also be used in the modeling process. If *channels* were used to model links in an ATM network, for instance, a link failure would be modeled by advancing the clock on the corresponding channel. The cause of the failure can come either from the source ATM switch or from the destination or from somewhere else. It makes sense in this case to be able to advance the *channel* clock from other places besides the source LP. However, this introduces some inconveniences as explained in the following.

Because of the asynchronous nature of the execution, a number of messages could be in the *channel* waiting to be processed at the destination. When the source *LP* advances the *channel* clock the kernel assures that the previous events stored in the *channel* are still processed at the destination.

If the destination *LP* is allowed to advance the clock, things get more complicated. The main reason is that in a simulation the destination *LP* of a *channel* cannot limit the source

LP for scheduling events anytime in the future. As a result, when the destination *LP* decides to advance the *channel* clock, in the *channel* there might be events already scheduled by the source LP. Simply discarding these events might lead to inaccurate results (if a link is down in an ATM network, cells are re-routed). Rolling back the source *LP* is also impossible due to the conservative character of the synchronisation mechanism in TasKit. Although other mechanisms to solve this problem might be possible, the solution, instead of simplifying the model, would complicate it.

To conclude this section, it is preferable to not allow the destination *LP* or any other *LP* or class in the model, except the sender to advance the time of the input channel. Modeling link failures or any other similar situation can be solved at the application level using other approaches.

5.1.1.3 Channel Class Interface

The issues regarding the implementation of *channel* class discussed so far suggests that the sender *LP* is the most suitable for controlling all the *channel* attributes and functionality including its creation. This can be achieved by making the class that implements the *LP* in TasKit (`sk_node` class) the only class that is capable of creating a *channel* (its output channel) and accessing its attributes. In this way, the kernel can ensure that changes and operations on attributes of a *channel* are performed by its source LP. Figure 16 illustrates the basic methods that can be implemented in the *LP* class (`sk_node`) that allow controlling the parameters of its output *channels*:

sk_node

void create_channel(sk_node *to, sk_time delay);
void create_channel(int lp_id, sk_time delay);
void open_channel(sk_channel *which, sk_time time);
void advance_clock(sk_channel *which, sk_time time);
void set_chdelay(sk_channel *which, sk_time time);
sk_time get_chdelay(sk_channel *which);

Figure 16 Part of the public interface `sk_node` class responsible for the management of a LP's output *channels*

The `create_channel` methods are the only way of creating *channels*. The source *LP* links itself with a destination *LP* given either by a pointer (`* to`) or by a number id

(`lp_id`) by calling this method. The other argument of the method is optional and represents the *channel* delay. If no delay argument is passed to the constructor, it assumes that the parameter is zero. The `sk_node` class has also methods to return references to its output channels as presented later in Section 5.1.2.4.

Note that the status of a *channel* is by default set to *closed*. In order to open a *channel* the `open_channel(sk_channel *which, sk_time time)` has to be called. For the reasons mentioned earlier, this method can be called only during the initialisation phase of the simulation (SimKit phase 2 and 3). If the second argument is passed to the method, it represents the value the *channel* time is advance to, otherwise this value is considered to be zero, i.e. the clock in the *channel* is not advanced.

It is not clear what the initial state of a *channel* should be. On one hand, for general applications in which only a subset of all the resources in the application is used the alternative presented above makes sense: it is more convenient to open only those *channels* that are needed instead of closing all the others. This is the case for the ATM-TN simulator. Only those *channels* that are going to be used in a particular experiment are opened at the beginning of the simulation. On the other hand, for an application with no unused *channels*, having the default status closed and explicitly opening each *channel* is somehow annoying and error prone. To support this alternative the constructor has to take another parameter, which is the value of the *channel* clock, and a `close` method should replace the `open` method.

The `advance_clock` method advances the *channel* time to the specified value. It checks the validity of the parameter before performing the operation on the *channel* clock and it can be called in the application in any phase of the experiment. The last two methods `set_chdelay` and `get_chdelay` can also be used for setting and respectively reading the delay of an output channel. The first method can be called only in the initialisation phase while the second one can be used in any phase of the experiment.

A consequence of this approach is that `sk_channel` class has no public interface. All the above operations can only be performed through the source *LP* of a channel. This

alternative allows the kernel to check each time if the operations on *channels* are performed by the source LP, that is, if *channel* argument (`which`) is one of the output *channels* of the *LP* object that called the method.

5.1.2 Logical Processes (LPs)

The `sk_lp` class in the previous SimKit versions is used to implement *LPs* as defined by *LP* modeling view. With the TasKit kernel *LPs* have the same functionality but additionally they also represent the vertices in the interaction graph of the application. Therefore, *LPs* in TasKit have to consider their connectivity and use *channels* when sending events to each other.

This is the main reason the `sk_lp` class was extended to the `sk_node` class to represent *channel* based *LPs* in the extended version of SimKit. The `sk_node` class inherits all the functionality of `sk_lp` class (it is a subclass of `sk_lp`) and, in addition, it implements specific features that allows it to work with *channels*. As seen in the previous sections, these features includes creation of *channels*, setting their status to opened or closed, advancing the clocks on the output *channels* etc.

Typically, an *LP* consists of an arbitrary number of input and output *channels* (Figure 15). Besides, an *LP* also maintains a local Future Event List (FEL) to store its *self-events*. *Self-events* are events scheduled by an *LP* to itself as opposed to events scheduled for other *LPs*, hereafter refer to as *external events or messages*. Conceptually, in the optimised synchronisation algorithm presented earlier, an LP's FEL can be seen as an additional input channel.

Another particularity of the *channel based LP* is that it belongs to a task. Grouping *LPs* into *tasks* has little impact on modeling but might have a great impact on application performance.

A number of issues regarding *channel based LPs* are discussed in the following sections.

5.1.2.1 Simplicity versus Efficiency

The *LP* execution mechanism in TasKit has been briefly introduced in section 5.1.1.2. Based on this algorithm, an *LP* repeatedly compares the external events on the non-empty input *channels* with the next self event in its FEL and selects the one with the lowest timestamp to process it. This requires a number of comparisons among the event timestamps usually proportional with the number of input *channels* and might be time consuming.

The optimised version of the above mechanism works slightly different to reduce the number of comparisons. First, an event from each input *channel* is inserted in the FEL in timestamp order. The *LP* execution consists of processing events from the FEL. Each time an event from the head of the FEL is processed its type is checked first. If the event was self scheduled then the actions associated with this event are completed. Otherwise, it is an external event and after its execution is completed, the next event form the same input *channel* is removed and inserted in the FEL. Comparisons are now performed only when inserting the next event in the FEL.

This mechanism, while efficient for *LPs* with many input *channels*, is not very suitable for *LPs* with one input channel. This situation requires only a comparison between the timestamps of the next external event and the next self event in order to determine which one has to be processed. In contrast, using the optimised implementation, this assumes inserting the next event from the input *channel* in the FEL and might consist of an arbitrary number of comparisons.

A similar situation was encountered with operations on the output *channels*. In order to take advantage and apply the most efficient scheme for each of these cases, four types of *LPs* corresponding to each situation should be used instead of one as follows:

- Single Input - Single Output (SISO)
- Multiple Input - Single Output (MISO)
- Single Input - Multiple Output (SIMO)
- Multiple Input - Multiple Output (MIMO)

An even more efficient way would be to further separately implement the source *LPs* (with no input *channels*) and sink *LPs* (with no output *channels*). The only difference among these classes is in the scheme implemented for event scheduling.

Although very efficient, this approach has a negative impact in the usability of the SimKit framework. First it increases the number of hot spots (different class for each type of LP). The programmer has to learn the difference between these hot spots in order to be able to efficiently use them. Since the only differences refer to the underlying implementation of the contracts, this approach defeats the main purpose of the framework of hiding the complexity of the underlying mechanism. It also can lead to code duplications. A `sk_node` class in the application, for example, represents an *LP* in the model that can receive/send from/to an arbitrary number of other entities. However, with the above approach, the same code has to be implemented in four classes one for each possible case: SISO, MISO, SIMO and MIMO. Besides the explosion in the number of classes in the system, this also results in code duplication.

The other alternative is to have one single *LP* class that represents all the above types of *LPs*. This class can use algorithms that are general enough but less efficient, as opposed to optimised ones for each type of LP. Alternatively, the optimised algorithms can be dynamically chosen at run time in each case based on the number of input and output *channels* of each instance of the class. The Strategy pattern (Gamma et al., 1996) is a good candidate for this latter option (Figure 17).

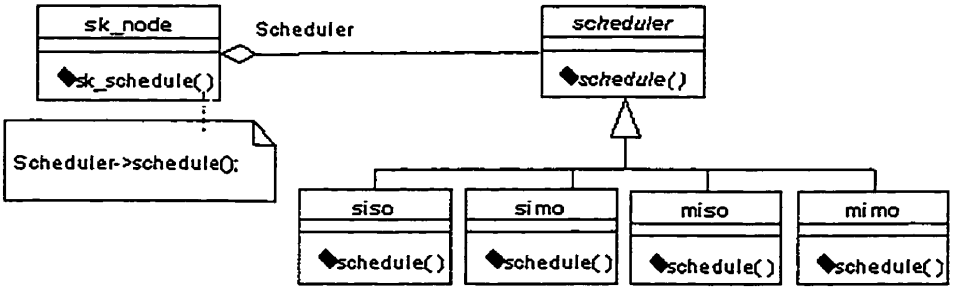


Figure 17 The configuration of `sk_node` class when using the Strategy design pattern

In this design approach, the abstract class scheduler is the base class for all the possible event-scheduling strategies. Its `schedule` method is implemented differently in its subclasses according to the number of input and output *channels*[6]. When an *LP* object is created the information regarding the number of input and output *channels* is passed to its constructor. Based on this information the *LP* object is automatically configured with one of these strategy classes by making Scheduler to refer to one of the concrete implementations. Therefore when the *LP* is executed, the most efficient scheduling implementation is automatically used. The advantage of this design is that the user needs to know nothing about the strategies. There are many variations and ways to implement a strategy pattern (see Gamma's book for more details (1996)). The `Scheduler` reference, for example, can be a pointer to an object or to a method.

Whichever approach is chosen, the option with only one *LP* class (`sk_node`) has the advantage of a simple and clear interface. Reducing the amount of code that users must write by minimising the number of classes that must be derived as well as the number of member methods that must be overridden, has been suggested by other OOF practitioners (Adair, 1995b). Because an efficiency price has to be paid, at the end a trade-off is made by choosing the solution that best meets the requirements of a particular problem domain.

5.1.2.2 Channels Identified by Pointers or Ids?

A *channel* is created to link a source *LP* with a destination. During the simulation experiment, the source sends events through that *channel* in order to be received by the destination LP. The issue here is whether to allow the user to select the output *channel* of an *LP* using a *channel* id (simple integer) or using a pointer.

Using a pointer is more efficient because there is no need to look for the *channel* that matches a specified id. However, some problems might appear in this case with *channels* that connect *LPs* within a cluster task. These issues will be discussed later when *tasks* are described.

---

[6] The assumption made here is that `schedule` method efficiently implements input and output operations.

5.1.2.3 Sending Events to Channels

There are two main types of events in TasKit: self-events and external events. Due to reasons explained later (see section event cancellation) scheduling each of these types of events is implemented by a separate method. These methods can be part of `sk_event` class or part of the `sk_node` class and discussing these two alternatives is the focus of this section.

There are two advantages when these methods are part of the `sk_event` class. First is compatibility (the `send_and_delete` method in the previous version of SimKit is also part of `sk_event` class). Second, they can be called from any other class simply by creating an event object and calling these methods on that object. When included in the `sk_node` class, these methods can be called only from `sk_node`'s subclasses or a reference to such an object has to be provided. On the other hand there are some advantages when the `sk_node` class implement these methods as discussed in the following.

The `schedule_event` method is used to schedule self-events. A self event is queued in the FEL of the *LP* that scheduled it. Therefore, in order to know where to send the self-event, the kernel has to know the sender LP. As a result, if `schedule_event` is part of `sk_node` class, its implementation is very simple and straightforward: the self-event is only inserted in the FEL of the LP. If the method is part of `sk_event` class, the `schedule_event` method has to take an extra argument (pointer to an LP) that identifies the *LP* that sent the message so that the event can be inserted in its FEL.

External events can only be sent through *channels*. To enforce this, the `send_and_delete` method in the old SimKit is replaced by `send_to_channel` method. The two arguments of the latter method are an identifier of the *channel* that is used for sending the event and the timestamp of the message. Since the destination of the event is identified by the *channel* argument no information about the source *LP* that sent the event is required. As a consequence, `send_to_channel` has the same implementation whether it is a method in the `sk_event` class or in the `sk_node` class.

An advantage when the method is part of the `sk_node` class is that the kernel can check and signal if the *channel* argument is not one of the LP's output *channels*.

To conclude this section, in general, implementing the event scheduling methods in the event class allows a more flexible design of an application while making them part of the *LP* class is less error prone.

5.1.2.4 The Public Interface of the sk_node Class

Following the above discussion a possible interface for the `sk_node` class is presented in Figure 18. The constructor takes two arguments representing the maximum number of input and output *channels* this *LP* uses. It performs initialisations such as the LVT, the number of events received so far etc. If its implementation uses the Strategy pattern discussed earlier, the constructor also configures the object with one of the event scheduling implementations according to its type (number of input and output *channels*).

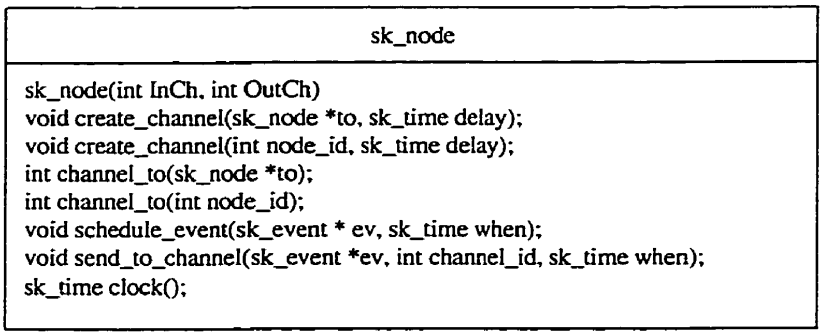


Figure 18 The possible public interface for `sk_node` class

There are two possibilities for creating a channel. Both `create_channel` methods perform the same type of action. The only difference is that one of them takes a pointer to the destination *LP* as a parameter while the other one takes the *LP* id. It is a similar situation for the two variations of `channel_to` method that returns the id of the *channel* to the specified *LP* destination. This method returns -1 if the *channel* has not been already created.

In the case that the event scheduling methods are part of the `sk_node` class they can have the interface presented in Figure 18. They take as arguments the event to be sent and the

timestamp of that event. Besides, `send_to_channel` also accepts an extra parameter representing the *channel* id that the event is sent through.

Finally, the last method in `sk_node` class returns the LVT for this LP.

5.1.3 Tasks

Closely related *LPs* can be statically grouped in *tasks* to be scheduled as a single unit. *Tasks* do not have a correspondence in a model as their existence is only for the purpose of efficiency in *LP* scheduling. They do not have state variables and contain no computation. However, the *task* concept is very important in CCT since *tasks* capture the patterns of *LP* connectivity and interaction.

The basic rules for grouping *LPs* into *tasks*, usually referred to as *task* partitioning, in order to obtain an improvement in performance are (Xiao et el., 1999):

- The *task* should contain enough *LPs* that its granularity minimises processor contention at the *task* queue.

- The *task* should contain *LPs* that generate events for each other in order to increase the cache locality and the amount of work done when a *task* is scheduled for execution.

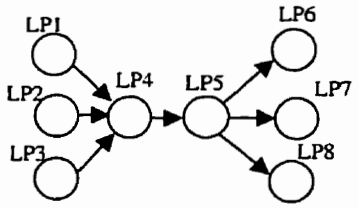
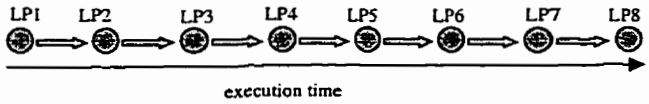
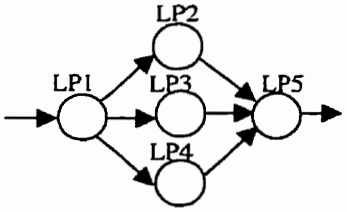
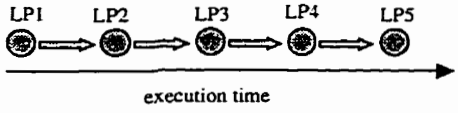
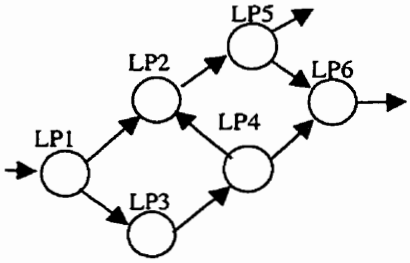
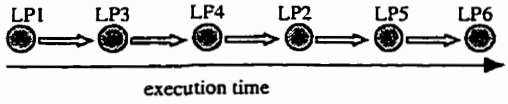
Currently, there are two types of *tasks*: *pipe tasks* and *cluster tasks*. The main difference between them is in the way they schedule their member *LPs*.

The *LP* scheduling algorithm in a *pipe task* takes advantage of the interaction configuration of their *LPs* in an attempt to maximise the execution time window of each LP. This as shown in a previous section can lead to an improvement of the performance of the system. For some *LP* particular interaction patterns, the causality relationships introduced by *channels* allow implementing efficient *LP* scheduling mechanisms. This is the case for *LP* groups that can form a logical pipeline. This enables a processor to execute one *LP* after another along the pipeline without explicitly scheduling each LP. Some examples or such possible configurations are illustrated in Table 4. *LPs* are scheduled to be executed

according to their causality relationships (see Section 5.1.1). In the first example, because there are no *channels* connecting the first three *LPs*, there is no causality relationship among them. Consequently, these *LPs* can be scheduled in an arbitrary order (LP1, LP2 and then LP3 – for instance) but before LP4. After LP4, LP5 is scheduled and at the end LP6, LP7 and LP8 again in an arbitrary order because there are no *channels* to link them.

Because this algorithm attempts to maximise the time window for each LP, it is more likely to have a larger number of events per execution session for each LP. However the algorithm does not guarantee this since it is possible that when the message population is low, the entire execution along the pipeline consists only of advancing the LVT of the *LPs*.

Table 4 Examples of *LP* interaction patterns with their causality relationships forming a pipeline.

Physical structure (*LPs* and *channels*)	Causality relationship forms a logical pipe line
	
	
	

Cluster tasks are used to group *LPs* with a configuration that does not match any of the *pipe task* configuration patterns supported by the TasKit kernel such as topologies with low lookahead cycles. An alternative solution to these cases could be to consider each *LP* as a *task* but this results in small granularity *tasks* that increase the contention accessing the shared task queue.

To avoid this situation, *cluster tasks* are used to group these *LPs*. The *LP* scheduling algorithm in a *cluster task* is performed according to the event with the lowest timestamp in that task. The execution is very similar to that of a simple sequential SimKit with all the *LPs* in the *task* sharing a single event queue (FEL). Because the synchronisation mechanism within a *cluster task* is based on a global event list, *cluster tasks* can be used to relief some of the constraints imposed by the conservative algorithm such as zero delay cycles. Figure 19 for example shows a very simple example with a zero delay loop between *LP* 2 and 3. It is the common case of an *LP* that requires feedback from another. This loop can cause deadlocks with the conservative synchronisation algorithms. Grouping the two *LPs* (or even all four *LPs*) into a *cluster task* eliminates this problem.

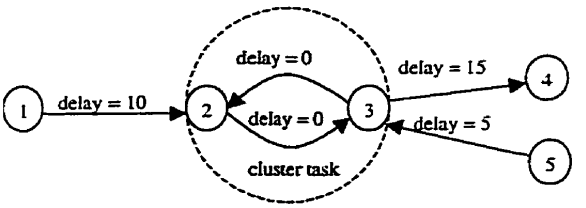


Figure 19 Example of a typical situation in which a *cluster task* can be used to avoid deadlock situations

A consequence of the fact that the synchronisation algorithm within a *cluster task* is based on central shared event list, there is no need for *channels*. Still in order to be consistent, *LPs* in *cluster task* still have to be connected through *channels*. However, internally, the kernel replaces the physical *channel* objects with a virtual *channel* to the FEL shared by all the *LPs* within a cluster task.

This generates the problem with *channels* identified by their pointers mentioned in Section 5.1.2.2: when two *LPs* are connected, a *channel* has to be created. The *channel* object is

removed if the two *LPs* are grouped in a cluster task. If *channels* are identified by pointers, any attempt to use a pointer within a *cluster task* for sending events causes fatal errors.

5.1.3.1 LP Configuration in Pipe-task

TasKit is still in its infancy. Although, there might be many possible *LP* configurations that can be grouped in a *task* to be more efficiently executed, optimised scheduling schemes are currently implemented only for those groups of *LPs* that are connected in a pipeline configuration. The generic pattern of such a pipe structure is outlined in Figure 20.

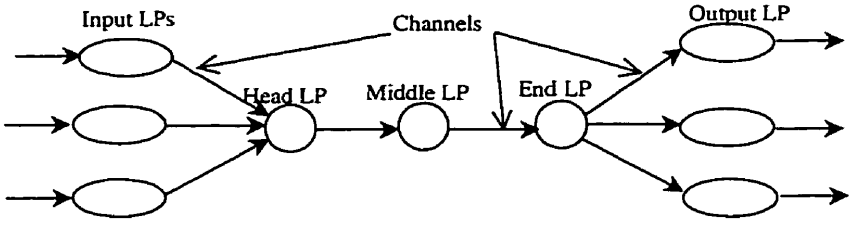


Figure 20 *Pipe task* generic structure

Typically, it is composed of a multiplexer *LP* (the Head *LP* in Figure 14) with multiple input *channels*, following by a pipe structure and ending with a de-multiplexer *LP* (End LP) with multiple outputs. The Head *LP* in a *task* is the merging point from a number of Input *LPs* but it has only a single output. After a number of one-input-one-output Middle *LPs* the pipe can end with an End *LP* that have single input and multiple outputs.

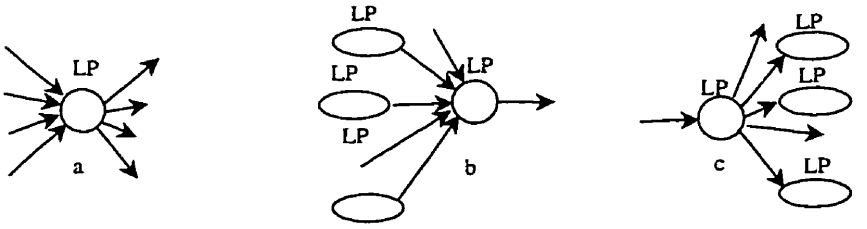


Figure 21 Other *task* patterns

Variations of this structure are also supported. For example, the Output, the Middle, the End and some input *LPs* from the generic structure can be missing (Figure 21b). Similarly, the variation with no Input, Head, Middle and some output *LPs* is also accepted (Figure 21c). As a result the simplest structure is the one presented in Figure 21a with only one

multiple input-output LP. All the other combinations of the generic pipeline structure are also supported.

5.1.3.2 Cluster Tasks versus Nodes.

Another way of grouping entities that appear in low lookahead cycles that has been investigated is by using *nodes*. The idea originates from the similarities that exist between the *channel* based *LP* in TasKit and a cluster task. Both have a number of input and output *channels*, a FEL and the same event scheduling algorithm based on the lowest timestamp event.

All these facts suggest that in TasKit there are two types of components. *LPs*[7] are used for receiving, processing and sending events (as in the previous SimKit version). *Node* is a new component that is used exclusively for scheduling events to their destination *LPs*. A *node* would have a number of input and output *channels*, a FEL and a number of simple *LPs* in a low lookahead cycle (Figure 22). These *nodes* map the vertices in the model's interaction graph. They eliminate the need for *cluster tasks* because vertices in a low lookahead cycle can be collapsed in a single vertex modeled by a *node* with multiple *LP* components. A *node* can only be executed sequentially and therefore it can be seen as the basic parallel unit in the model. Note that the problem with *LPs* in low lookahead cycles can be also solved by modeling these *LPs* as one single large LP.

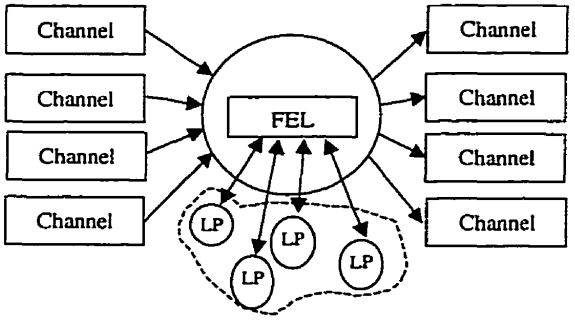


Figure 22 The structure of a *node*

[7] These are simple *LPs* as described in the LP modeling view.

The main advantage of such an approach is that there is no need to create *channels* among the *LPs* within a *node* (in fact this is impossible since entities in a *node* are simple *LPs*). Therefore, the kinds of problems generated by *channels* identified by a pointer within a *cluster task* do not apply in this case.

However, a major difficulty with this approach is that *LPs*, in order to communicate to other *LPs* in other *nodes*, use *channels* provided by their *node*. A connection from a *node* to another is through a channel. Therefore, if two *LPs* inside a *node* want to send messages to an *LP* at the other *node*, they have to share a channel. Unfortunately, the order of messages in a shared *channel* cannot be guaranteed to be non-decreasing and this can lead to causality errors. There are a number of solutions to solve this problem. One of them is to duplicate a *channel* between two *nodes* to avoid it being shared. Another alternative would be to implement a mechanism that ensures a non-decreasing order of events in a shared channel. A simple analysis of these alternatives reveals that the complexity of the API increases for the first solution or the efficiency price tag is too big for the second one.

Considering all these, the solution using the *cluster task* remains the preferred one.

5.1.3.3 Global Variables within a Cluster Task

In order to maintain a minimal process syncronization in PDES, the *LP* modeling requires that PPs be disjoint, that is, *LPs* that represent them cannot share attributes. Although usually this is not a severe restriction, it may be burdensome in some situations. *Cluster tasks*, on the other hand, are executed sequentially and therefore banning global variables within a *cluster task* is not necessary.

Moreover, global variables in *cluster tasks* can be sometimes very useful. It is the example of a wireless cellular *network model* in which each mobile station is modeled as an LP. Each *LP* belonging to the same base station shares a number of global data such as the number of radio *channels* available, their frequencies, etc. By allowing shared attributes within *cluster tasks* the problem becomes relatively easy to model and implement: *LPs* that belong to a base station are grouped in a *cluster task* so that can share the common attributes.

Global data in *cluster tasks* can be supported and might be very useful in many situations. The reverse effect is that the users might be tempted to use this solution excessively. As a result, the execution is forced to be sequential even when there is parallelism available in the model.

5.1.3.4 The Public Interface for the Task Class

Since *tasks* have no correspondent in the model, they are not used in an application directly. The only exception is at the beginning when *LPs* are first grouped to create the *tasks*. Because grouping *LPs* in *tasks* is performed based on communication patterns, even this operation can be automated in future releases. Consequently, *task* class has a very simple public interface consisting of a small number of access methods. Because the user does not work with *task* objects directly it is more appropriate to make task's access methods reachable only through `sk_node` class. The interface of `sk_node` class is extended in this case with a number of methods (Figure 23).

The `registry` method is very important. It adds the *LP* to a *task* of specified type and identifier. These *tasks* if not existing yet are automatically created by the kernel. Although the *LP* scheduling algorithm in *pipe-task* is based on *LP* topology, the kernel does not require a particular order in which LPs are registered.

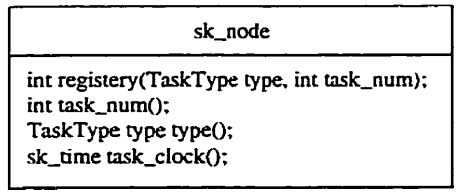


Figure 23 The public interface of `sk_task` class

The access methods return the *task* id, *task* type or its clock. They are used especially for debugging purposes.

5.2 The copy_and_delete Method

The optimisation introduced by the `copy_and_delete` method refers to the mechanism of creating and deleting event objects. SimKit applications are typical complex and time consuming. A very large number of events are generated during an experiment and

therefore their creation and deletion count as an important percentage of the execution time.

The mechanism of allocating and de-allocating event objects has been already optimised in SimKit. The kernel allocates at the beginning a buffer of memory of size multiple of the maximum of the sizes of all the subclasses of `sk_event` in the system. Previously, the application has to inform the kernel about the sizes of all the classes derived from `sk_event` class. The `new` and `delete` operators are overridden in `sk_event` class so that, whenever the application requests a new event object the kernel returns a free spot from the buffer pool and marks it as in use. When an event object is removed its corresponding spot in the buffer pool is marked as free. This mechanism is efficient because it does not call system functions for allocating and de-allocating memory space every time an event object is created or destroyed. However, the memory is not used very efficiently (a memory spot of maximum event size is allocated for all event objects including those that are smaller in size).

The WarpKit kernel requires every event to be state saved before it is processed to enable rollbacks. Therefore, its memory location cannot be re-used until the event is fossil collected. This mechanism is used with the other kernels, although the memory location for the current event is not state saved and becomes available after its handler has been called. Reusing the memory spot of the current event for the next event once it becomes available eliminates the need for searching for another one and can save on execution time. Moreover, since in many of the applications, a message block is just reused and all info it contains remains unchanged, it is also not necessary in this situations to initialize the fields of the reused message.

This is the goal of the `copy_and_delete` method. The method is called when the application requires a memory spot for a new event object. The WarpKit implementation of this method simply returns a pointer to a new memory location (as the new operator). With the other kernels, it returns a pointer to the location of the current event object to be reused. Note that the `process` method provides a pointer to constant for the current

event. Therefore the current event cannot be modified unless the constness is casted away, that is, another pointer to non-constant object for the same location is obtained from the `copy_and_delete` method. After calling `copy_and_delete` the location of the current event object can be initialised with data corresponding to the new event object so the old information can be lost. In order to prevent memory leaks, removing an event object is still performed by the kernel if `copy_and_delete` has not been called in the `process` method. The kernel also makes sure that the `copy_and_delete` method is not used more than once inside a `process` method (there is only one memory location to be reused for each call of the process method). A failure to do so generates an error. The mechanism of `copy_and_delete` method is illustrated in Figure 24.

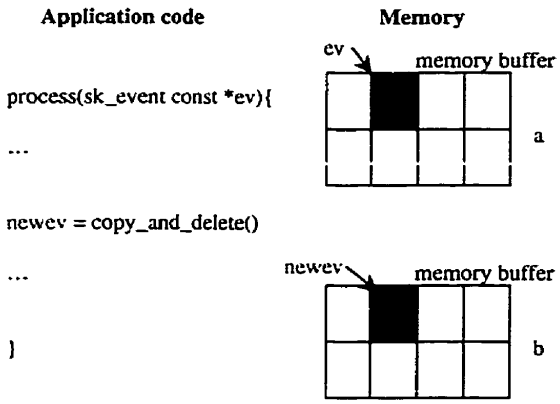


Figure 24 The effect of using `copy_and_delete` method on the memory buffer. a. the memory buffer before `copy_and_delete` call. b. the memory buffer after.

The `copy_and_delete` method can be used in all the four testing examples. The ATM example is very suitable for this feature because most of the events in the system represent ATM cells. In switches modeled by LPs, the received cell is forwarded further with almost no modifications. Therefore because the current event and the next one describe the same ATM cell, the memory location of the current event can be reused without even being reinitialised.

This new feature can improve the performance of the application up to 10% in some cases as the preliminary tests show. However, it implies working at a low level with memory allocation and requires extra care. Before using it, this mechanism should be very well

understood in order to prevent errors. If performance is not a big issue the application implementation can be simplified by not using this feature.

5.3 Self-Event Cancellation Mechanism

Event cancellation refers to cancelling events that have been already sent out by an *LP* but have not occurred yet[8] (the LVT at which the cancellation takes place must be less than or equal to the receive timestamp of the message being cancelled). DES applications often utilize this ability. In the Ethernet model for example, a station *LP* model sends the start of a framework if the line is quiet. In the same time it schedules a self-event to signal the end of transmitting the framework. However, if a collision is detected the transmission of the framework stops and future attempts to send it are scheduled. The initial self-event is meaningless in this case and the application has to disregard it once received. The implementation without a cancellation mechanism necessitates a counter to distinguish the self-events that are outdated (their corresponding framework transmission has been postponed) from the real one and this complicates the user's implementation. With the self-event cancellation mechanism, one has only to cancel the self-event once the collision is detected. Note that in the process view model this corresponds to a situation in which a method that simulates the passage of time in a process is interrupted by an external event or process.

While it is possible to simulate activities without the ability to cancel events, the availability of such a mechanism often simplifies the model of an application. However, its implementation requires different degrees of complexity for different types of kernels.

In a sequential implementation based on a central event list, this feature is relatively straightforward to implement; the desired event is simply located and removed from the FEL. Due to its capability of rolling back an LP, the TimeWarp mechanism also enables the implementation of the event cancellation mechanism (Lomow, Das and Fujimoto,

1991). The implementation of this feature becomes more difficult in the conservative kernel. Because no rollbacks are allowed, the kernel can dispatch an event that might be cancelled in the future to its destination *LP* only when there is no possibility of being cancelled.

An alternative to event cancellation is self-event cancellation. Self-event cancellation refers to the possibility of cancelling self-events. It has two main advantages: it is relative simple to implement within all the kernels, sequential or parallel, and it is the most frequently used. For self-event cancellation, the *LP* that requests the cancellation coincides with the destination LP. It is therefore not possible that this event has been already processed and there are no rollbacks or causality errors. In all the parallel kernels (WaiKit, WarpKit and TasKit) the implementation of this feature consists only of removing the self-event from the FEL of the *LP* or cluster task. The sequential implementation is similar to one for general event cancellation mechanism as presented earlier. Self-event cancellation not only that introduces no performance penalties but also might increase the efficiency by avoiding calling the event handler for cancelled events.

Besides the implementation aspects, from the application point of view, this type of cancellation is probably the most frequently used. Self-events are usually sent by an *LP* to remind itself the end or the beginning of an activity. Sometimes a situation that requires the activity to interrupt or to be postponed can occur. The corresponding self-event has to be cancelled in this situation as in the example of the Ethernet application.

Because the self-event cancellation is bounded to an *LP* (only the *LP* that sends the event can cancel it), it make sense to implement this mechanism in the class. There is one method that does this and it is added to the public interface of the `sk_lp` class:

```
int cancel_event(sk_event *ev);
```

---

[8] This type of event cancellation is sometimes referred to as event retraction in literature to avoid the confusion with the special meaning of the term "cancellation" in the Time Warp mechanism.

The `cancel_event` method cancels the self-event `ev` (`ev` must be sent by `schedule_event` method). It returns 1 if successful otherwise it returns 0.

5.4 Design Guidance Rules for SimKit Libraries

The section briefly discusses some of the issues related to the implementation of a number of common elements that can be reused in many applications. In DES, these elements include random number generators, different types of queues, statistic gathering and interpreting tools, experiment managing tools and input/output data handling tools. Although very often used in simulation applications, these components are not mandatory and they are not part of the framework basic mechanism. Therefore it is more appropriate to implement these them in libraries that can be selectively used for each application rather then as another layer on top of SimKit. There are several advantages associated with this approach: better performance less memory usage (an application uses only those elements that are needed), simplicity (the API is not "polluted" with all the possible features), etc.

The new utility libraries are part of the framework and therefore they also have to meet SimKit's requirements presented in the previous chapter. Since performance is very important for SimKit, the design of the libraries has been strongly influenced by this requirement. Some of the main design guidance rules that have been followed especially for those elements that are supposed to be very often used in a simulation (random number generators, statistics and queues) are:

- *Avoid dynamical allocation and deletion of objects.* This is one costly operation that was avoided as much as possible. When it was necessary, a special mechanism similar to that for event object allocation and de-allocation was used. This implies that the library code takes the responsibility of managing the memory for these objects. The drawback is a possible inefficient memory usage.

- *Store the related data allocated in contiguous memory areas.* This is simply achieved by grouping the data inside an object. There are two main advantages. First, since the data is related, it is very likely that many operations will use it. If the data is spread

across the memory, loading it from different locations takes longer and these operations are slowed down. Second, contiguous memory locations are easier to state-save for rollbacks in the optimistic approach.

- *Avoid pointers.* Pointers, beside the extra level of indirection associated with them, encourage the use of non-contiguous memory locations.

- *Avoid virtual methods.* Virtual methods introduce an extra level of indirection (Meyers, 1996). This can cause delays in the system especially if the specified method is very often called during the execution (it is situated along the critical path of the execution).

- *Avoid templates.* This rule pertains more with the compatibility requirement. Some of the compilers do not support all the operations with templates. Besides, some of the SimKit users might not be familiar with templates.

There are also other C++ specific techniques that can be considered for increasing the efficiency of the implementation (Meyer, 1996). Most of the rules mentioned above are in contradiction with the methods employed by the OO programming. In fact, inheritance, because it has little effect on performance, was the only OO technique that has been extensively used in implementing the libraries. As a result, code duplication could not be avoided and this affects other properties of the software such as maintainability, extensibility and others. Nevertheless, the code implemented in the libraries is not to be changed so often and therefore these drawbacks are not as important as the overall performance of the system.

Currently, three libraries have been implemented for the SimKit framework. Some others are planned to be introduced in the future. Due to the proprietary nature of the product described in these sections, the discussion is presented at a high conceptual level.

5.4.1 Random Number Generators

Random number generators (RNGs) are routines used to generate a stream of numbers that have a random distribution. They allow stochastic or probabilistic effects to be

incorporated into a simulation. Examples of such situations include random sources, random execution times, random priorities for different elements, etc. The statistical results of running the simulation are highly dependent of the random number generator. Because RNGs use deterministic algorithms, the numbers they generate are not random and therefore they are more accurately refer to as pseudo random number generators.

A random variable can assume only a subset of values. The probability distribution is defined as the values a random variable can assume and the corresponding probabilities of each. There are many different probability distributions that can model different situations. All the testing examples need random number generators with different probability distributions.

The design of the RNG is very simple: the main RNG is the parent class for all the classes in the library. Each probability distribution class specialises the RNG parent class. It can add different other parameters of its own, parameters that characterises different types of distributions such as mean, interval limits etc. As a result of the design presented above, parameters for each distribution probability are part of an object and allocated in contiguous memory locations with all the advantages derived from this.

5.4.2 Queue Libraries

Queues are one of the most commonly used elements in a simulation model. Almost every simulation application uses queues for storing events or other types of objects. All testing examples for instance require queues: the ATM applications for storing ATM cells, the Ethernet example for storing frames while the line is busy and finally the Qnet model for storing messages with low priority until it is safe to send them.

The design of the queue library is not trivial. It has to be very flexible in order to maximise its reuse on a broad range or situations. Thus ideally, a queue should be used not only for storing objects of `sk_event` type but also different other types of objects. Queue objects are easy to integrate with the SimKit framework and its tools. The library is also flexible allowing different queue policies to be used. While the basic queue policies such as First-

In-First-Out (FIFO) or First-In-Last-Out (FILO) are offered in the library, the design also allows other new ones to be easily added.

5.4.3 Statistics Utilities

The ultimate goal of building a simulation is to obtain information about the system being under investigation, information that is used in making decisions. Data gathered from running the simulation application, usually refer to as the statistics, is difficult to be interpret unless it is processed. Data processing consists of computing a number of metrics that statistically characterises the data such as mean, standard deviation, maximum, minimum, and others. These parameters offer a more accessible description of the system's behavior than the raw data itself. Statistics can be un-weighted sample statistics or time weighted sample statistics depending whether the time component is taken or not into consideration.

Another possibility of analysing the data is to represent it in graphical form. Charts, diagrams and histograms can provide a more meaningful feedback for interpretation.

Statistics library has been design as a collection of statistics classes. A statistics object can be associated with the element of the model that is of interest and samples are collected at specified points in the simulation time. At the end, the statistics object computes and returns a number of metrics that describe its corresponding element including mean, standard deviation, second and third momentum, confidence interval etc.

5.5 Debugging Features

The control flow in a framework is different from other types of applications (Section 3.3). One of consequences associated with this is that debugging SimKit applications might become very laborious and difficult.

A preventive alternative would be to perform lots of checks during the execution in order to detect any possible error caused by misusing the framework's features. However, these

checking operations would be time consuming and useful only during the debugging process. In order to avoid this, two compiled versions of the framework can be offered:

1. An extended version of the framework compiled with debugging facilities that is able to follow the application execution step by step. The kernel displays different messages about the status of the simulation, its phases, current actions and different other variables of interest. It also performs different checking operations in an attempt to prevent some of the possible coding or modeling errors. The user can identify where the error occurred and he/she can take measures to solve the problem. This extended version of the framework is used only in the debugging phase because it is slower.

2. After the debugging phase a version without these debugging features is used instead. The main functionality of the application is the same but the results are obtained faster as no checks are performed. The two versions of the framework can be easily implemented and compiled using debugging flags in C++.

This approach can be used with any software program. Besides this, DES applications offer another method for debugging especially the correctness of the model: event tracing. Event tracing consists of marking some of the events in the application as special. Receiving such events leads to different tracing related actions such as: checking different conditions, displaying the status of different elements in the applications (LPs, *channels*, etc.). Events caused by a trace event can also be marked as trace events up to a specified level. Thus, the domino effect caused by a trace event can be followed and the behavior can be compared with the theoretical one.

5.6 Summary

This chapter has discussed some of the main issues related to new functionality implemented in the SimKit framework. Two are the directions that have been identified for improving the framework: new abstractions due to changes in the problem domain and new common elements identified in many applications.

In the first category, issues related to new abstractions due to changes in the problem domain are discussed. *Network models* is the new problem domain and it enables the implementation of a more efficient simulation engine, TasKit. Some of the possible alternatives for the implementation of the three new TasKit elements in SimKit - *channels*, *channel* based LPs and *tasks* - have been analysed. These solutions imply trade-off decisions that have to be made. The `copy_and_delete` method is another abstraction intended for improving SimKit's performance. Its mechanism and the rationale behind it were presented. The last new abstraction, the cancellation mechanism is a feature designed mainly to facilitate the modeling of problem. It also can improve the performance of the system.

In the second part of the chapter, two main categories of new common elements have been briefly discussed. They consist of utility libraries that are not part of the kernel but can be used with it and tools for debugging support. Based on the analysis of the alternatives, a number of guidance rules for the design of very efficient library classes have been identified and presented.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

This chapter evaluates the work presented in this thesis and concludes with a number of possible future research directions. It presents an evaluation both of the extended SimKit framework and of the approach followed to meet the goal of the thesis.

In the first part (Section 6.1), the extended SimKit framework is evaluated with respect to how its new added features meet the SimKit requirements formulated in Chapter 4. Based on this and on the objectives stated in Chapter 1, an evaluation of the entire work is presented in Section 6.2. In the last part of the chapter, some conclusions are drawn (Section 6.3) and a number of directions for future research work are suggested (Section 6.4).

6.1 Evaluation of the Extended SimKit Framework

This first part of the chapter is concerned with an evaluation of the extended SimKit framework. There were two main causes that led to the changes in the extended SimKit framework: new abstractions due to a new problem domain (*network model* applications) and new common components found in many applications. The main goals of these changes were: first, to improve SimKit's performance and second to make it more usable. The analysis presented in the previous chapter was focused on the design of the API level, the implementation details being either too complex for the objective of this thesis as in the case of the TasKit kernel or protected by the proprietary nature of some of the SimKit's new components. While the main concern throughout the design process was how to implement the proposed features in a very effective way, attention was also paid to modeling and usability issues.

A first evaluation of the extended SimKit is performed in terms of the OOF key terms presented in Section 4.3.1 (Table 5).

Table 5 The OOF key terms in the extended SimKit

Key term	The extended SimKit framework
Set of classes	Besides the existing classes, the extended SimKit version introduces a number of new classes corresponding to the new components in the *network model* domain. Some of these new components includes classes to *model channels, tasks, channel based LPs.*
Design	The same overall structure of an application is defined in the extended SimKit. However, the main difference is that the component interaction in the system has to be described statically by building channels that are exclusively used during the experiment for exchanging messages.
Abstract design	The extended SimKit's design is also abstract due to the fact that `sk_node` class is abstract.
Solutions for a family of related problems	SimKit offers solutions only for *network model* problems. This is a sub-domain of the general DES problem domain.
Reuse & Reuse Granularity	The extended SimKit framework also promotes reuse at all the three granularity levels: class level, design level and analysis level. An improvement of the reuse at the class level is the result of using the built-in components from the accompanying libraries.
Usability	Some of the usability factors such as performance and adaptability have increased in the extended SimKit for the *network model* domain. Others such as robustness and simplicity have not changed much. However, for problems other than *network models* most of these factors are poor and the usability suffers.

A more detailed evaluation of the extended SimKit is performed based on the SimKit's requirements identified in section 4.1:

1. Efficient event-oriented *logical process* view of simulation
2. Transparency with respect to the underlying simulation kernel (compatibility)
3. Embedded in a general-purpose language that supports a wide range of applications
4. Ease of use

These are high level requirements and, consequently, some of them are ambiguous. It makes SimKit difficult to evaluate against such requirements and this is one of the limitations of the evaluation as discussed in the following sections.

6.1.1 Efficiency of the Extended SimKit Framework

As any PDES application, efficiency was the main requirement for SimKit and the incentive for many of the changes discussed in the previous chapter. Therefore, a great deal of attention was paid to this aspect when different design and implementation alternatives were analysed.

The concurrent execution of a simulation application on multiple processor computers is highly sensitive to many factors including some that are unrelated with the actual application model. Changes as small as the memory alignment, for example, can modify the workloads of the processors causing the execution to follow a totally different path. As a result, oftentimes, it is very difficult to find a correct correlation between changes in the input data and the efficiency of the execution. For the same reasons, it is also difficult to get accurate measurements when executing the application in parallel.

Things get even more complicated when estimating the performance of a PDES simulation framework such as SimKit. Other application dependent factors such as the parallelism of an application or its granularity make measuring the performance of the framework more difficult. This is mainly because the overhead introduced by the framework depends on these factors (e.g the task queue contention depends on the granularity). Different benchmarks are used in this case for comparing different design alternatives, but the results might vary for other applications.

These are the reasons why it was difficult to measure and prove the performance of the extended SimKit framework. However, a number of experiments with the ATM-TN simulator benchmark have been performed to illustrate rather than to prove the performance implications of the extended framework. Table 6 presents the speedup

obtained with the TasKit kernel over WarpKit, the second most efficient kernel as results from Figure 12.

Table 6 Speedup of obtained with the TasKit kernel relative to the WarpKit kernel

Num proc.	Wnet-1	Wnet-3	NTN-0	NTN-1	NTN-2	NTN-3
1	4.7	4.1	4.1	4.4	4.7	5.1
2	5.3	5.9	3.9	4.8	5.1	5.2
4	3.8	3.6	3.5	4.7	5.6	5.5
8	4.0	2.6	2.6	4.4	5.0	4.7
12	3.5	2.2	2.2	4.5	5.0	4.9
16	3.3	2.5	1.8	4.3	4.9	4.9

Wnet and NTN model two different ATM network topologies: the topology of a regional ATM test-bed network in western Canada (11 ATM switches) and the topology of a Canada-wide experimental network called the National Test Network (54 ATM switches) (Unger et al., 1999). Upon each network model, different traffic mixes are used to produce network traffic loads ranging from light to medium to heavy. Wnet-1 and Wnet-2, for example, have different numbers of sources of different types: 12 sources in Wnet-1 and 21 in Wnet-2. The model of Wnet-1 consists of 181 *LP*s grouped in 26 tasks and generates $22*10^6$ events for the simulation of 10 seconds of real time while Wnet-3 consists in 173 *LP*s grouped in 12 tasks and generates $18*10^6$ events for simulating the same amount of real time. NTN-0, NTN-1, NTN-2 and NTN-3 represent increasing traffic loads for the NTN topology. NTN-0 has 99 different traffic sources and its corresponding model consists of 869 *LP*s (112 tasks). It generates $48*10^6$ events for the simulation of 5 seconds of real time. NTN-1, NTN-2 and NTN-3 have the same number and deployment of traffic sources (355) but increasing traffic load. As a result their corresponding models consist of the same number of *LP*s (1381) grouped in 112 tasks but generating increasing number of events for the simulation of 5 seconds of real time ($73*10^6$ events for NTN-1, $132*10^6$ events for NTN-2 and $216*10^6$ events for NTN-1). Table 6 represents the results of running the same experiments in parallel on 1, 2, 4, 8, 12 and 16 processors on a SGI

Power Challenge machine. These results clearly show that for these two models and scenarios, TasKit is between 1.8 and 5.8 faster than WarpKit.

Another new feature, with probably a smaller impact on SimKit efficiency is the `copy_and_delete` method. Preliminary results with the three examples show an approximate 10% improvement in performance when this method is used. However, this estimate is not generally applicable. The actual performance gain for a specific model depends on many factors such as: the granularity of the application, the size of the initial buffer pool, the size of the FEL etc. When running in parallel, things get even more complicated, because `copy_and_delete` can change the workload on the processors. The self-event cancelation mechanism is also supposed to speed up the execution, as the `process` method in `sk_node` is not called for a cancelled event and the length of the FEL decreases. However there is a cost associated with finding and deleting the cancelled event from the FEL.

These are changes in SimKit's core design and have the biggest impact on its performance. Although it is difficult to give a quantitative estimation of the efficiency of the new extensions to SimKit, their impact is likely to be quite significant for a large variety of *network model* applications.

6.1.2 Ease of Use

Easy to use is the second most important requirement for SimKit and probably the most ambiguous. As this is a high level non-quantifiable requirement, the analysis and the evaluation of different design alternatives from this perspective cannot be but subjective. Although this bias has been reduced by using the three examples approach and involving simulation experts in the design process, it could not be completely eliminated. Another obstacle was that TasKit, being a new kernel, has not been extensively used in applications and there was no appropriate feedback from users. Therefore, a possible solution is to consider getting users to test the prototype of the extended version of SimKit and provide feedback before lunching the final version

There are a number of factors that affect this requirement such as learning curve, application development effort, validation and defect removal etc. Because these factors are more tangible, the evaluation of *easy to use* requirement consists in evaluating how these factors have been addressed.

- ***Learning curve*** can be improved by keeping the SimKit API as simple as possible. New components including new hotspots were chosen to be meaningful for the user and to guide the modeling of an application. The process of designing the API was iterative. The first extended SimKit prototypes were designed for maximizing the performances of the framework. The API of the first one consists, for example, of 7 new classes, 4 new hotspots introducing a total of 58 methods etc. With so many new elements, the effort required to understand SimKit and how to use its features increases. Besides, including too many components in the API might be error prone. When *channel* objects can be directly accessed from an application, for example, they can be misused (see section 5.1.1.2 on advancing the clock in channel). Therefore, numerous design alternatives that could simplify the API were considered and analysed. In the end, the simplest solution consists in only one new class (`sk_node`) with a total of 18 methods. This class extends and replaces one of hotspots in the previous version of the framework (the `sk_lp` class). This final API is not only very simple but also has little negative impact on the performance of the system and it also avoids some possible errors that might be introduced in a model (e.g. an *LP* cannot send to an *LP* that it is not connected to).

  Another way to improve the learning curve is to provide good documentation for the SimKit framework. The discussion presented in the previous chapter can be used to understand the new features added to the kernel, the rationale behind the design decisions that were made and some of the resulting consequences. Also, the documentation of the new version of SimKit can be accompanied by the three concrete examples that have been used in this thesis (The ATM-TN example is too complex to be used for this purpose). Including concrete examples in documenting an OOF is one of the solutions proposed by framework researchers.

- ***Application development effort*** is directly correlated with the learning curve. An easy to learn and understand framework is more likely to be correctly used and reduces the required development effort. There are, however, other important factors that impact the development effort.

  An important factor is the reusability of components. Reusability is one of the main goals in software engineering that can greatly reduce the development effort. It is known that in most situations the reusability of the components facilitates the development of an application although it is difficult to evaluate this factor quantitatively. One metric that can reflect the impact of component reuse is the number of lines of code (LOC) in an application. Basically, each component that is reused (from a library for example) reduces the size of the application with its corresponding number of LOC. However, there is no direct correlation between LOC and the development effort. Some small size software can be more complex and difficult to implement and test compared to other that is very large but simple. One example in the first category is a random number distribution. Testing the randomness of a random number generator algorithm or distribution is often more difficult than testing other larger software components.

  In spite of the fact that it is difficult to quantitatively measure the effect of the new features in SimKit on the application development effort, keeping its API simple, documenting it and reusing more components from libraries are factors that contributed to facilitating the implementation of SimKit applications.

- ***Validation and defect removal***. Testing the software implemented by an application that uses the framework is an important aspect of the overall development effort. SimKit is easier to use from this perspective if it facilitates finding and removing defects in an application. This was in fact the main goal of offering debugging facilities in SimKit. Again, their effect on the development effort is believed to be beneficial but a concrete evaluation of this is hard to be performed.

6.1.3 Compatibility with Previous SimKit Versions

The compatibility requirement has been extended so that it includes not only compatibility among different kernels in SimKit but also compatibility with the previous version of the framework. The compatibility of each of the new components that has been added to the extended framework is evaluated bellow.

- ***Support for TasKit engine***. With the introduction of the new elements, *channels*, *tasks* and *channel based* LPs, the compatibility with the previous version of SimKit becomes difficult to achieve. LPs, for example, cannot send messages directly among themselves, but through *channels* using the `send_to_channel` method. This means that old applications cannot be executed with TasKit unless major changes (*channel based LPs*, *channels* and *tasks* creations) are made in the application and this defeats the whole purpose of compatibility.

  In order to overcome this problem, the original `send_and_delete` method that sends messages directly to the destination *LP*, is still maintained in the extended version of SimKit. However, it allows sending events to any other *LP* and the conservative synchronization algorithm cannot support this functionality. Therefore `send_and_delete` can be used only during those phases of the experiment that are executed sequentially (see phases of a SimKit experiment in section 4.2), that is, prior to phase 4 (Simulation Execution).

Therefore, a new phase has been introduced between *LP* Initialization and Simulation Execution. This phase (Phase 4 in Table 7) is executed sequentially based on a central event list. All events that use `send_and_delete` method are sent and received during this phase while events that use `send_to_channel` are received only in the Simulation Execution phase. This phase ends when the central FEL becomes empty and the parallel execution based on the CCT algorithm starts afterwards. There are two main advantages of this solution:

Table 7 The execution phase of the extended SimKit

Phase	Name	Description	Execution
1	Program Initialization	Construction of static objects	Sequential
2	SimKit and Model Global Initialization	Initialization of the model and *LP* creation	Sequential
3	*Logical process* Initialization	Initialization of *LPs*	Sequential
4	Sequential Simulation execution	Sequential execution. *LP* uses `send_and_delete` method to directly exchange and process messages.	Sequential
5	Parallel Simulation Execution	Parallel execution of the model using TasKit kernel	Parallel
6	*Logical process* termination	*LP* termination operations	Sequential
7	Simulation Clean-up	Simulation termination specific operations	Sequential

1. ***The compatibility with the previous versions of SimKit*** can be more easily achieved. An entire simulation experiment can be run sequentially during this new phase (e.g. all events in the experiment are sent and received). If no channel based implementation is provided, the application completes its execution in this phase and skips phase 5 of the simulation. It does not require in this case that *channels* and *tasks* be created.

2. ***The possibility of implementing complex initialisation operations***. There are cases in which the initialisation of the model is a complex operation. *LPs* in the model have to exchange information to each other in order to set up their initial state. This is the case of the ATM-TN simulator. *LPs* representing ATM switches exchange information among themselves in order to initialize routing tables and other initial states. In this phase, although the simulation time does not advance, events can be directly sent to *LPs* in the system and processed. Without being able to send messages directly to an *LP* these initialization operations would be more difficult to model and implement.

- The *copy_and_delete* method is designed to optimise the creation of event objects for the sequential and the two conservative kernels. It returns a pointer to the memory location of the current event to be reused. With the optimistic kernel, however, this location has to be state saved and cannot be used for a new event object. Therefore this method cannot have the same functionality with WarpKit engine. To solve this compatibility problem, the `copy_and_delete` method has a different implementation with the WarpKit engine so that instead of returning a pointer to the location of the current event object, it returns a pointer to a new location. It has the same effect as the `new` operator. Because the `copy_and_delete` method can be optionally used, there are no compatibility problems with the old applications that do not use it.

- Self-event cancellation mechanism, similarly to `copy_and_delete` method, is an optional mechanism. Therefore, there are no compatibility problems with old applications that do not use it. Also, one of the main reasons the self-event cancellation mechanism has been preferred to a general event cancellation mechanism is that it can be easily implemented within any of the existing kernels. As a result, different configurations of the SimKit framework are compatible from this perspective.

- Libraries, more than any of the above new components, have been design to be optionally used with any of the provided kernels. Therefore they are compatible with any other SimKit application.

6.1.4 Embedded in a General-Purpose Language that Encompasses a Wide Range of Applications

C++ is also the preferred programming language for implementing the new features in the new version of SimKit.

6.2 Evaluation of the Thesis

The primary goal of this thesis as described in chapter 1 was:

" ... to identify and investigate some of the main issues related to extending an existing discrete event simulation package to target the specific problem domain of *network models* by using object oriented framework techniques. The emphasis is placed on issues that concern the design of the application programmer interface (API)."

In order to meet this goal, a number of objectives have been identified in chapter 1 to be fulfilled:

1. To study discrete event simulation, sequential and parallel, and to frame an in depth understanding of the main issues in this field.

2. To study object-oriented frameworks and to get insight into this field.

3. To study an existing discrete event simulation framework, SimKit. This includes identifying its initial requirements and understanding its mechanism and implementation.

4. To identify directions in which SimKit could be extended and to choose an approach for performing these changes.

5. To analyse some of the issues related to the implementation of the changes in the extended SimKit and to discuss alternatives for addressing these issues.

6. To draw some conclusions from this process in a form of "lessons learned" and to propose further development and research directions based on the experience and insights gained.

6.2.1 Fulfilment of the Objectives

The above objectives have been fulfilled as follows:

Chapter 2 made a review of the discrete event simulation field. It covered all the main concepts in simulation highlighting those that specifically refer to discrete event simulation. The following concepts have been described here:

- The definition of simulation and its importance;
- The main phases of a simulation experiment;
- The difference between discrete and continuous simulations;
- The four modeling views: *event view, process view, activity view and logical process view*;
- The sequential mechanism for implementing a *LP* view of DES;
- The need for parallel discrete event simulation and the two main classical synchronisation algorithms: conservative and optimistic;
- The new TasKit synchronisation algorithm.

Chapter 3 surveyed the field of object-oriented frameworks and concepts related to it such as those that refer to object oriented programming. The following concepts have described here:

- Object oriented programming concepts, characteristic attributes and benefits of using OO
- Object oriented frameworks definition and particularities
- Main components of OOFs
- The process of developing OOFs
- The main strengths of using OOFs
- The main weaknesses of using OOFs

Chapter 4 introduced SimKit simulation tool. The presentation of SimKit contained its initial requirements, a description of its API and the main phases of a simulation that uses

it. SimKit was then identified as an OOF for DES problem domain and based on this fact, an analysis of the SimKit was conducted. Also, the need for extended the SimKit and the cause of the changes were identified and presented. The "three examples" approach is used to extend SimKit. The four models that have been used for this purpose were presented (Appendix A).

Chapter 5 presented the main issues related with upgrading SimKit. Modifications to SimKit are identified at the beginning and categorised upon their primarily objectives as follows:

- New abstractions due to changes in the problem domain: TasKit and the modification it requires to the SimKit API, the `copy_and_delete` method and the self-event cancellation mechanism.

- New common elements found in SimKit applications: libraries and debugging facilities.

Each of the changes was analysed first mainly with respect of its efficiency and usability perspective. Consequences of applying different alternatives were presented and a possible implementation of the API was suggested.

Chapter 2 and chapter 3 fulfil the first two objectives. Objective 3 is fulfilled in chapter 4 subsections 4.1 and 4.2. and so is Objective 4 through the remaining of the chapter. Chapter 5 fulfilled objective 5. Finally, objective 6 is going to be fulfilled in the subsection 6.2 of this chapter.

6.2.2 Fulfilment of the Goal

Based on the above facts it can be concluded that the goal of the thesis was successfully fulfilled. First, a number of directions for upgrading SimKit have been identified based on user requests, kernel optimisations, OOF experience or other simulation packages. Based on this, concrete new features that can be integrated in the framework have been analysed and possible solutions have been suggested. These suggestions have been supported with

examples of applications implemented using SimKit. The focus of the discussion was on describing the mechanism for the new features and their representations at the API level rather than on the complexity of the underlying implementation. OOF techniques have been used in the identification of the necessary upgrades, their implementation and testing.

6.3 Lessons Learned

During the research process a number of other interesting aspects have been observed:

- ***Difficulty of the process of designing and implementing an OOF***. This is one of the issues that most OOF practitioners agree on (Johnson and Russo, 1991), (Gamma et al., 1995) and others. DES frameworks do not make an exception from this fact. Although benefiting from simulation expertise and using one of the approaches suggested in the OOF literature, the process of designing the upgrades to SimKit has been long and difficult. The main difficulty resides in making a decision on selecting the final version from different design alternatives. Because most of the time trade-off solutions have to be considered it is necessary to have as much knowledge and expertise in the problem domain as possible in order to make the right decision. Getting feedback from the users is desirable in this case.

- ***The problem domain of a framework***. The problem domain is a very important factor in building a successful OOF. If the problem domain is too narrow, the reusability of the framework is limited. If the problem domain is too large, as in SimKit's case, the framework is intended to cover a large variety of specific problems. Its features are general enough so that they can be reused by all the applications in the problem domain but require a great deal of effort in order to be specialised for different particular applications. A number of new features could not be implemented in the extended version of SimKit because they were not general enough for the given problem domain. A possible solution for this issue might be to address the problem domains in hierarchical fashion and have a hierarchy of frameworks as presented in the next section.

Another important consequence, particularly for DES field, of restricting the problem domain of a framework is that it allows that the most efficient algorithms for that specific domain to be used. There is a great potential for obtaining very good performance for *network model* applications when using TasKit. However, although *network models* is a broad domain, it does not include all the applications supported by the previous version of SimKit. In other words, the extended SimKit with its new API actually restricts the problem domain. Applications that vary the interaction pattern among LPs and possibly the number of LPs during the simulation can be executed with the old SimKit but have a very small chance of success with the extended version.

- ***Object-oriented techniques applicability***. In spite of all the advantages associated with OO programming, OO paradigm is not a panacea in all the circumstances. The general belief that structured programming should be used for better performance and OO programming for better quality (modularity, maintainability, decoupling etc.) is also not always applicable. Although OO specific operations are time consuming, there are cases in which OO solutions outperform their structured programming alternatives in terms of efficiency. The conclusion is that a problem should be analysed before deciding on the technique that is more appropriate to be used or what features of that technique can be used. There are situations, for instance, where only a subset of the OO techniques can be successfully applied. It is the case for SimKit's library where out of all the OO features inheritance is used the most. In other cases, if impossible to replace, OO time consuming operations can be optimised as in the case of creation and deletion of events objects. It is desirable therefore that, especially for applications with different constraints (e.g. time constraints), the advantages brought by OO in the quality of the resulting software to be balanced by its weaknesses in such efficiency.

6.4 Future Research Directions

The work presented in this thesis is aimed at exploring the possibility of extending an existing DES tool using OOF techniques. Based on the experience and knowledge gained

during this work a number of questions have been raised. They lead to a number of general areas that show promise for further research.

Further exploration of the CCT algorithm.

TasKit is at its infant stage. The preliminary results show that further research is required in this area. This includes optimisations of the algorithm and different other improvements. One of them is the automation of the *task* partition process. Since the *task* partition is based on the network static structure, this activity can be automated. This will simplify the API by eliminating the need for the task grouping.

On the other hand, as Johnson and Russo noticed (1991, p. 32) "a framework should not be used widely until it has proven itself, because the more widely it is used, the more expensive is to change it". A framework is proven itself only after it has been used. Therefore, a final version of the framework should be considered only after receiving feedback from the users. It is desirable to have t a "beta version" of the framework based on which users can provide feedback. Changes can afterwards be implemented in the API and in the framework accordingly.

Breaking down the problem domain.

One of the main challenges with the actual SimKit framework is that it addresses DES in general, which is a very large problem domain. In order to use SimKit, a great deal of work is required to specialise it. Reducing this effort is possible only by having more of the low level components implemented by the framework, that is, by specialising the framework. These two requirements are contradictory: a specialised framework cannot be general enough and vice versa.

A solution for mitigating this problem might be to consider a hierarchy of frameworks. Frameworks at the top of the hierarchy are intended for a broad problem domain such as the DES domain. These frameworks offer general simulation support that consists of basic simulation components such as: a mechanism for sending and receiving events, tools for random number generation, statistics collection, queuing, etc. SimKit can be included in

this category. At a lower level, frameworks specialise the top one(s) in more specific domains: telecommunications, manufacturing, combat simulations etc. In this way they can implement more functionality. A framework design for telecommunication applications, for example, can include the implementation of different switch prototypes, link types and telecommunication protocols. A framework for military simulation will implement other types of components. This hierarchy can continue with other lower levels; the telecommunication framework is specialised for ATM networks, IP networking protocols, etc (Figure 25). It is more likely that following this approach, frameworks at the lowest level reach the final stages the framework evolution presented by Roberts and Johnson (1996) becoming a visual language so they do not require any programming. The existing ATM-TN simulator represents an example in this sense. It is a black-box framework (all the components - switches, traffic sources etc are provided) built on top of a white-box framework (SimKit).

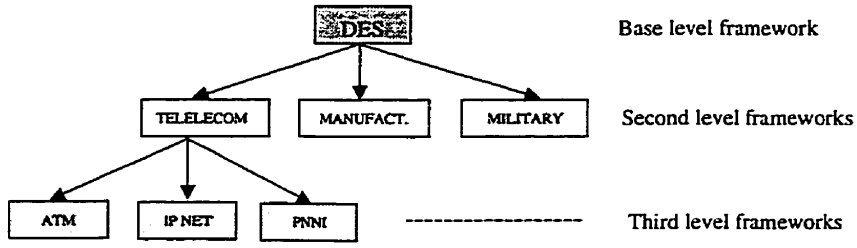


Figure 25 Framework hierarchy

This approach has also been suggested by other OOF practitioners (Adair, 1995b):

> "One approach is to build a very flexible, general framework form which you derive additional frameworks for narrower problem domains. These additional frameworks provide the default behavior and built-in functionality, while the general framework provides the flexibility".

Another advantage of this approach is in selecting the right kernel. Currently there are four simulation kernels SimKit can be configured with: one sequential and three for parallel execution. Studies showed that each of these kernels is more suitable for particular types of problems. Therefore, the user can select any of them depending on the application. However, at the lower level, this might not be necessary. If TasKit is proved to be suitable

for telecommunication applications because it facilitates the modeling and execution of *network model* applications, then it is the default (maybe the single) simulation kernel for the telecommunication domain framework. The low level framework can then fully exploit TasKit's capabilities without being constrained by conditions imposed by other kernels.

Use the experience of applying OOF techniques to the DES domain in other domains.

This work has been concentrated on applying OOF techniques to a specific problem domain (DES). The experience and the knowledge gained by OOF community by using these techniques in different fields has been applied and has been used in the DES field. It is believed that the OOF research community can also benefit from this experience. DES is a relative mature domain with specific requirements such as efficiency constraints. Innovative ideas and knowledge with PDES frameworks such as having the hotspots and the framework implementation in different programming languages as in TeD's case or configuring the framework with different abstractions or kernels as in SimKit can be further applied in other problem domains.

6.5 Summary

This chapter summarises the research that was performed in this thesis and has two main parts. In the first part the extensions made in SimKit are evaluated according to the requirements that have been presented in one of the previous chapters. The second part of the chapter shows how the objectives and goal of the research were met. Finally some of the main lessons learned during this work are discussed and based on these facts and future research directions are suggested. .

REFERENCES

Abadi, M. and Cardelli, L. (1996). **A Theory of Objects,** New York, Springer

Adair, D. (1995a). Building Object-Oriented Frameworks Part1, **AIXpert online**, February, http://www.developer.ibm.com/library/aixpert/feb95/feb95toc.html

Adair, D. (1995b). Building Object-Oriented Frameworks Part2, **AIXpert online**, February, http://www.developer.ibm.com/library/aixpert/may95/may95toc.html

Ball P., (1998), Introduction to Discrete Event Simulation, http://www.strath.ac.uk/Departments/DMEM/MSRG/simulate.html, November 1998.

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. and Zűllighoven, H. (1997). Framework Development for Large Systems, **Communications of the ACM 40**(10), October, pp. 52-59.

Booch, G. (1994). **Object-Oriented Analysis and Design**, Menlo Park Ca., Addison-Wesley.

Bosch, J., Molin, P., Mattson, M. and Bengtson, P. (1997). Object-Oriented Frameworks – Problems & Experiences, **Proceedings of the 23rd International Conference in Technology of Object-Oriented Languages and Systems, TOOLS '97** USA, Santa Barbara, California, July 28 - August 1, pp. 203-214 http://www.ide.hk-r.se/~michaelm/papers/ex-frame.ps

Brugali, D., Menga, G. and Aarsten, A. (1997). The Framework Life Span, **Communications of the ACM 40**(10), October, pp. 65-68.

Bryant, R. E. (1977). Simulation of Packet Communications Architecture Computer Systems, **MIT-LCS-TR-188**, Massachusetts Institute of Technology.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996). **A System of Patterns: Pattern-Oriented Software Architecture**. West Sussex, England, John Wiley & Sons. 1996.

Chandy M. K. and Misra J., (1979). Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, **IEEE Transactions on Software Engineering**, Vol. SE-5, No5, September.

Cleary J. and Tsai J. J., (1996). Conservative Parallel Simulation of ATM Networks, **Proceedings of the 10th Workshop on Parallel and Distributed Simulation**, pp. 30-38.

Das S. R., Fujimoto R. M., Panesar K., Allison D. and Hybinette M., (1994). GTW: A Time Warp System for Shared Memory Multiprocessors, **Proceedings of the 1994 Winter Simulation Conference**, December 11-14, Lake Buena Vista, Florida, pp. 1332-1339.

Demeyer, S., Meijler, T. D., Nierstrasz, O. and Steyaert, P. (1997). Design Guidelines for 'Tailorable' Frameworks, **Communications of the ACM 40**(10), October, pp. 60-64.

Doscher, D. and Hodges, R. (1997). Sematech's Experience with the CIM Framework, **Communications of the ACM,** 40(10), October, pp. 82-84.

Fayad, E. M. and Schmidt, D. C. (1997). Object-Oriented Application Frameworks, **Communications of the ACM,** 40(10), October, pp. 32-38.

Fujimoto R. M., (1990), Parallel Discrete Event Simulation, **Communication of the ACM**, No 10, Vol. 33, October, pp. 30-53.

Fujimoto R. M., 1993, Parallel and Distributed Discrete Event Simulation: Algorithms and Applications, **Proceedings of the 1993 Winter Simulation Conference**, pp.106-114.

Gamma, E., Helm, R., Johnson, H. and Vlissides, J. (1995). **Design Patterns – Elements of Reusable Object-Oriented Software**, Reading Mass., Addisson-Wesley

Gangopadhyay, D. and Mitra, S. (1995). Understanding Frameworks by Exploration of Exemplars, **Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95),** IEEE Computer Society Press, ISBN 0-8186-7078-9, July 1, pp. 90-99.

Gomes F., (1993). A Survey of GVT Algorithms, http://www.cpsc.ucalgary.ca/~gomes/PAPERS/GVT.ps

Gomes F., Franks S., Unger B., Xiao Z., Cleary J. and Covington A., (1995). SimKit: A High Performance Logical Process Simulation Class Library in C++, **Proceedings of the 1995 Winter Simulation Conference (WSC'95)**, Arlington, Virginia, pp. 706-713.

Gomes F., (1996). Optimising Incremental State Saving and Restoration, **PhD Dissertation**, University of Calgary, August 14th

Jefferson D. R., (1985). Virtual Time, **ACM Transactions on Programming Languages and Systems**, Vol. 7, No. 3, July 1985, pp. 404-425.

Johnson, R. E. and Foote B. (1988). Designing Reusable Classes, **The Journal of Object-Oriented Programming,** 1(2), pp 22-35, ftp://st.cs.uiuc.edu/pub/papers/frameworks/designing-reusable-classes.ps.

Johnson, R. E. and Russo, V. F. (1991), Reusing Object-Oriented Designs, University of Illinois tech report UIUCDCS 91-1696, ftp://st.cs.uiuc.edu/pub/papers/frameworks/reusable-oo-design.ps

Johnson, R. E. (1992). Documenting Frameworks Using Patterns, **Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages, and Applications OOPSLA '92,** pp. 63-76, ftp://st.cs.uiuc.edu/pub/patterns/papers/documenting-frameworks.ps

Landin, N. and Niklasson, A. (1995). Development of Object-Oriented Frameworks, **Master Thesis, CODEN:LUTEDX(TETS-5231)/1-146**, Department of Communication Systems, Lund University, http://www.tts.lth.se/Personal/bjornr/Papers/OOFW.ps

Lomow G., Das S. R. and Fujimoto R. M., (1991). Mechanism for User-Invoked Retraction of Events in Time Warp, **ACM Transactions on Modeling and Computer Simulation**, Vol.1, No.3, July 1991, pp. 219-243.

Maria A., (1997), Introduction to Modeling and Simulation, **Proceedings of the 1997 Winter Simulation Conference**, December 7-10, Atlanta, Georgia, pp. 7-13.

MedModel, (1998). PROMODEL Corporation, http://www.medmodel.com/products.html

Meyers S., (1996). **More Effective C++ - 35 New Ways to Improve Your Programs and Designs**, Addison-Wesley Publishing Company, Inc.

Miller J. A., Nair R., Zhang Z. and Zhao H. (1997). JSIM: A Java-Based Simulation and Animation Environment, **Proceedings of the 30th Annual Simulation Symposium**, Atlanta, Georgia, pp. 31-42

Mostafari, H. (1998). Performance Effects of Applying Design Patterns to a Telecom System - **Masters Thesis**, Department of Systems and Computer Engineering, Faulty of Engineering, Ottawa, Ontario, -submitted.

Opdyke, W. F. (1992). Refactoring Object-Oriented Frameworks, **PhD thesis**, University of Illinois at Urbana-Champaign, ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z

OPNET, (1998), MIL 3, Inc. · 3400 International Drive, NW · Washington, DC · 20008 http://www.mil3.com/products/modeler/home.html

Page E. H. and Nance R.E., (1994). Parallel Discrete Event Simulation: A Modeling Methodological Perspective, **Proceedings of the ACM 8th Workshop on Parallel and Distributed Simulation PADS94**, Edinburgh, Scotland, 6-8 July, pp. 88-93.

Perumalla K., Oglieski A. and Fujimoto R., (1996). MetaTeD: A Meta Language for Modeling Telecommunication Networks, **GIT-CC-96-32**, Technical Report, College of Computing, Georgia Institute of Technology.

Perumalla K. and Fujimoto R., (1996). A C++ Instance of TeD, **GIT-CC-96-33,** Technical Report, College of Computing, Georgia Institute of Technology.

Pooley, R. J., (1987). **An Introduction to Programming in Simula**, Blackwell Scientific Publications.

Posnak, E. J., Lavender, R. G. and Vin, H. M. (1997). An Adaptive Framework for Developing Multimedia Software Components, **Communications of the ACM**, 40(10), October, pp. 43-47.

Pree W. (1995). **Design Patterns for Object-Oriented Software Development**. Addison-Wesley, Reading Mass. 1995.

Roberts, D. and Johnson, R. (1996). Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, presented at **the Third Conference on Pattern Languages and Programming (PloP '96)**, Allerton Park, Illinois, Sept. 4-6, http://st-www.cs.uiuc.edu/users/droberts/evolve.html

Schmid, H. A. (1997). Systematic Framework Design by Generalization, **Communications of the ACM**, 40(10), October, pp. 48-51.

Shewchuk, J.P. & Chang, T.C. (1991). An approach to object-oriented discrete event simulation of manufacturing systems, **Proceedings Winter Simulation Conference**, Phoenix, Arizona, USA: IEEE: 302-311.

SimKit (1995). SimKit - Application Programmer's Interface, Version 1.1, **TeleSim Technical Report**, Jade Simulation International Corporation, December 14.

Taligent, (1996). Leveraging Object-Oriented Frameworks – A Technology Primer from Taligent, **Taligent White Papers,** http://www.ibm.com/java/education/ooleveraging/index.html

Tanenbaum A., (1996). **Computer Networks**, 3rd edition, Prentice Hall.

Unger B. W., Xiao Z. and Cleary J., (1999). High Performance Task-based Parallel Simulation of ATM Networks, submitted for publication.

Unger B.W., Covington A., Gburzynski P., Gomes F., Ono-Tesfaya T., Ramaswamy S., Williamson C. and Xiao Z. (1995). A High Fidelity ATM Traffic and Network Simulator, **Proceedings of the Winter Simulation Conference**, Washington D.C, December 1995, pp. 996-1003.

Williamson C., Unger B. W. and Xiao Z., (1998). Parallel Simulation of ATM Networks: Case Study and Lessons Learned, **Proceedings of the Second Canadian Conference on Broadband Researc (CCBR'98)**, Ottawa, Ontario, pp. 78-88.

Xiao Z. and Unger B. W., (1995). Report on WarpKit - Performance Study and Improvement, **Technical Report 98-628-19**, Computer Science Department, University of Calgary, May 1995.

Xiao Z., Unger B. W., Simmonds R., Cleary J. G., (1999). Scheduling Critical Channel in Conservative Parallel Discrete Event simulation, submitted to PADS99.

APPENDIX A

ATM-TN simulator

The ATM-TN (Asynchronous Transmission Mode – Traffic and Networking) is a cell level ATM simulator built at University of Calgary (Unger et. al., 1995). The simulator built on top of SimKit models in detailed different types of traffics and switches used in an ATM network. It uses real benchmarks for studying different aspects of ATM networking and traffic modeling.

The model of such a benchmark consists of a large number of *LPs*, usually in the order of hundreds and a very large number of events in the order of tens of millions. The model is very large and fairly detailed. Several types of sources are able to generate different types of traffic. A number of switch models that simulate real switches are also included and the ATM cells in the model have different service types.

Most of the events in the system simulate the arrival and departure of ATM cells. The amount of computation associated with processing such events is very low and mainly consists in operations such re-routing, queuing or eliminating the cell. This results in a very small granularity simulation that along with its fixed connectivity property makes the CCT approach particularly suitable for this application. The ATM-TN example is very important because, historically, CCT algorithm came as an attempt to solve the performance difficulties encounter with this type of application.

ATM example

The ATM example is much simpler model of an ATM network. There are no types of services as in a reality although cells can have two priorities. The model for a switch is illustrated in Figure 26.

A switch has four inputs and four outputs. The actual cell routing in a switch is performed by a central processing unit (CPU). It receives cells from the four inputs and directs them

to one of the outputs according to their destination addresses. There is no storage space (buffers) at inputs because it is supposed that the CPU speed is at least four times the transmission speed and all inputs are equally served. This means that there is no cell loss at the input. However, cell loss can be encountered at the output in spite of the fact that each output has a cell buffer of a specified size. Cells with higher priority are served first but there is no pre-emption, that is, a high priority cell cannot stop the current transmission of a low priority cell.

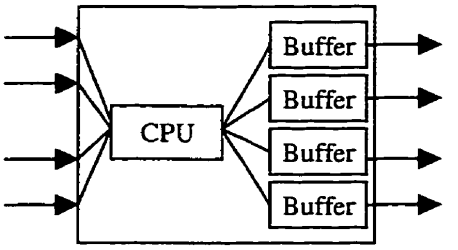


Figure 26 ATM switch

These switches are connected to form a symmetric network (Figure 27). First four switches are connected as in Figure 27a to form a cluster (the links are bi-directional) and four such clusters are connected through two backbones switches to create the whole network.

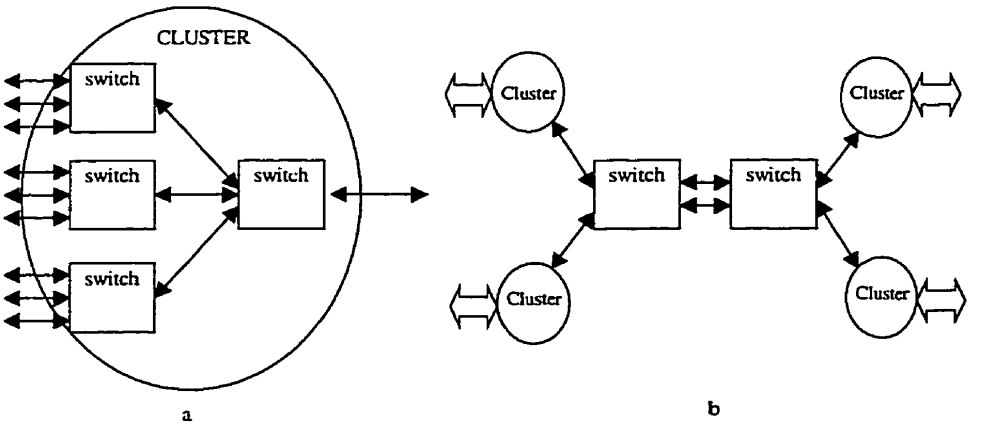


Figure 27 Switch networking. a Cluster structure. b Network structure

Cells of different priorities are pumped into the network randomly. The cell priority generator follows a normal distribution while the cell interarrival time has an exponential distribution. The means for these distributions as well as the length of output buffers in each switch are application parameters set by the user. The purpose of an experiment with this model may be to study the impact of the above input parameters on the network performance (cell loss, cell delay etc.).

A straightforward *LP* model of this problem is to have a switch modeled by five *LPs*. One *LP* represents the inputs and the CPU because the nature of the problem imposes the input and routing activities to be executed sequentially. The other four *LPs* representing each of the four outputs of a switch (Figure 28).

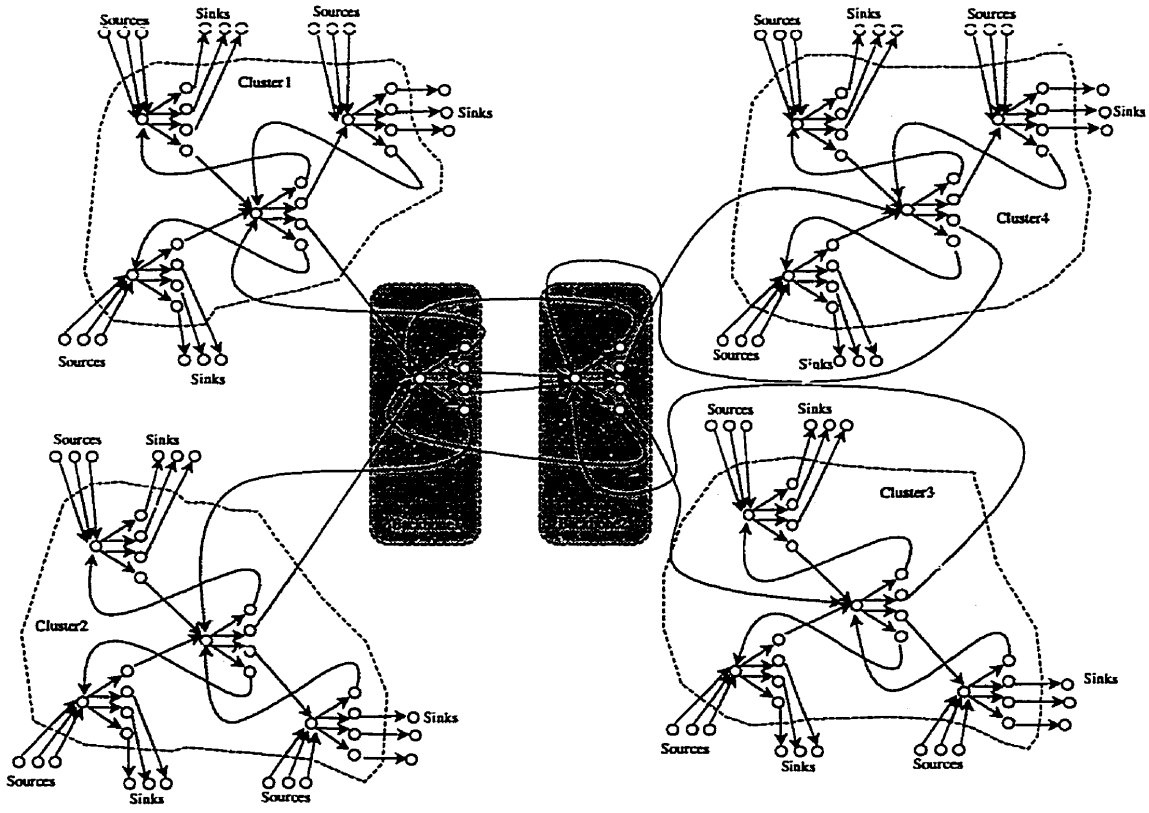


Figure 28 The *LP* model of the ATM problem

The Ethernet model

The Ethernet model is intended for studying the Ethernet protocol. The Ethernet protocol (Tanenbaum, 1996), the most popular Local Area Network (LAN) technology was developed by Digital, Intel and Xerox and has been standardized by the IEEE as 802.3. The protocol specifies that all workstations are connected to shared media and can transmit whenever they have data. The data to transmit is packed in frames of variable length. If a collision occurs both stations wait a random time defined by an algorithm called the backoff algorithm. After the waiting time elapses, if the line is quiet, the sender attempts to restart the transmission or the frame otherwise it listens the line and retransmits when the line becomes quiet.

A way of modeling such a problem is to have each station represented by an LP. This is arguably the best solution since a model in which an extra *LP* models the Ethernet (cable) itself might be more efficient (the number of events in the system can be reduced). Nevertheless, the main objective of modeling the problem is not to build the fastest version but to see how SimKit (both current and updated version) supports the implementation of a particular model that meets the requirements of a network application.

Each station has also a corresponding source that generates frames and a sink that receives frames that are addressed to this station (Figure 29).

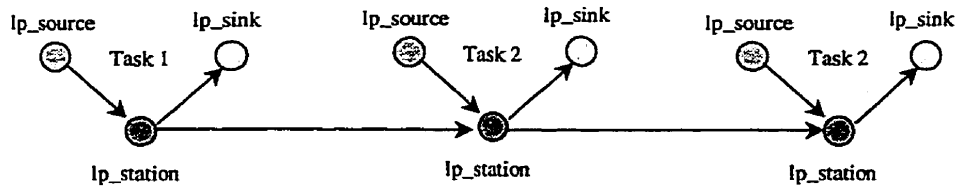


Figure 29 The model of a simple 3 stations LAN.

An `lp_station` object can generate six types of messages, some of each have no correspondent in the real Ethernet protocol. They are following presented. The first three types correspond to real messages in the protocol while the later three have no equivalent in the real world but are important for the model:

1. `StartM` represents the start of a new frame

2. `StopM` represents the end of a frame

3. `CollisionM` is emitted by the sender station when it detects is a collision in the network. Its presence also means that the sender stopped sending the current frame and rescheduled it later on.

4. `NewM` represents a new frame that has to be send by source to a station

5. `DoneM` signals a sender that the time necessary to send the current frame elapsed. In this case the sender sends a `StopM` message.

6. `WaitM` is used when a frame is resent because of a collision and it signals the sender that the random time in the backoff algorithm elapsed.

The `lp_station` class models a station connected in a network that uses Ethernet protocol to send and receive frames. Depending on the kind of messages a station receives it can be in one of the four states as follows:

- `Idle`: the station is not sending or receiving any frame and the line is quite.

- `Busy`: the station is receiving a frame or the line is busy, i.e. other stations communicate to each other.

- `Sending`: the station is currently sending a frame

- `Trying`: the station encountered at least one collision when attempting to send the current frame and it is trying to re-send it. This state is the most complex one and it has its own states:

    - `resending`: when a frame is re-sent after at least one collision.

    - `waiting`: a frame is waiting a random time defined by the backoff algorithm before attempting to send the frame again.

- `listening`: the line was busy after the random time elapsed and now the frame is listening the line. Once the line becomes quiet it will try to re-send the frame

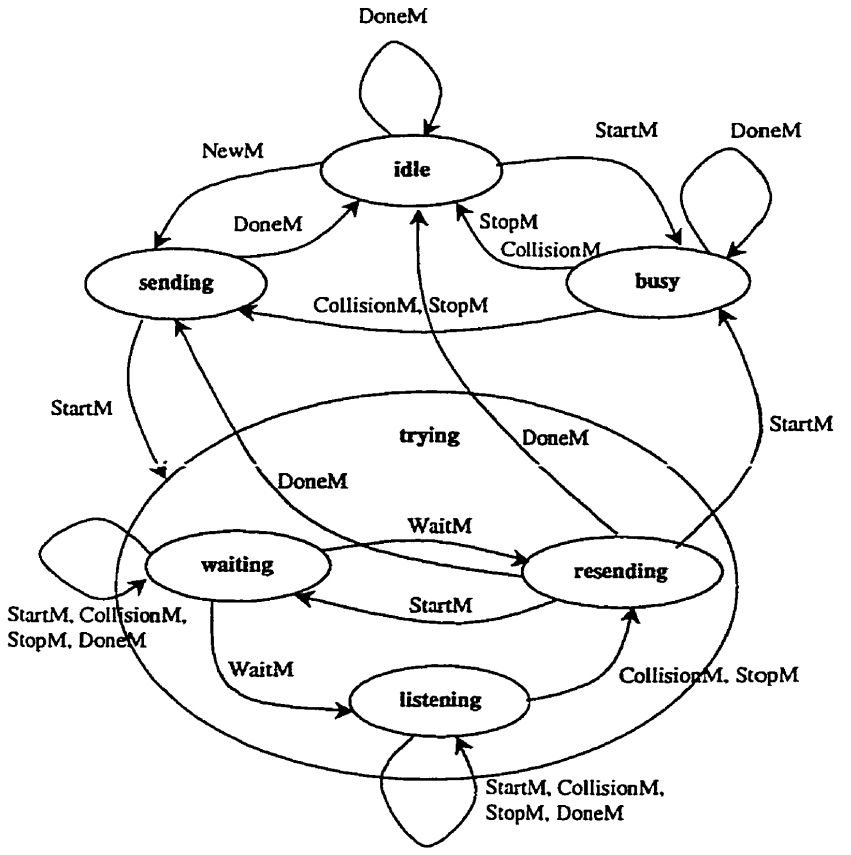


Figure 30 lp_station state transitions in the Ethernet simulation model.

The current state of an *LP* and the type of messages it receives dictate its next state. Figure 30 illustrates all the possible *LP* state transitions. From the `idle` state for example an *LP* can move either to `sending` state when it receives a `NewM` message or to `busy` after it processes a `StartM` message. Going back to the `idle` state can occur when a frame is transmitted after one (`sending` state) or many (`resending` state) attempts. Also, a `CollisionM` or a `StopM` message in the `busy` state can change the state of an *LP* to `idle`. `DoneM` do not modified the state of the LP. Actually `DoneM` this is a redundant message but this aspect will be discussed in the next chapter.

Although the diagram might suggest a fairly complex mechanism, the implementation of the functionality is not so difficult especially when uses the State design pattern (Gamma et al., 1995).

Qnet model

Qnet model is a simple classical *network model* often used to test the performance of different synchronization algorithms in PDES (Fujimoto, 1990). The model is composed of a number of similar *LPs* situated in the corners of a hypercube of a given dimension. Figure 31 shows such networks for hypercubes with two or three dimensions.

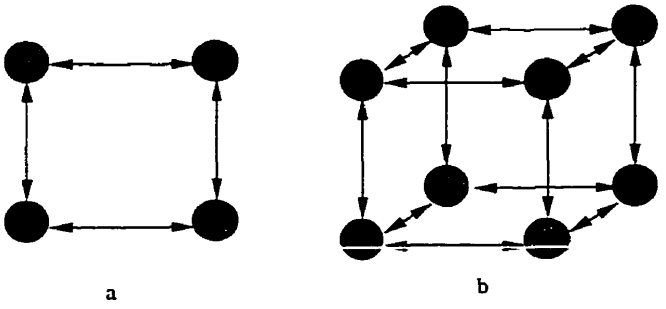


Figure 31 Examples of hypercube networks. a. dimension = 2. b. dimension = 3

An *LP* can receive and randomly forward messages from/to any of its neighbor *LPs*. The service time for each message is an exponential distribution random number with a given mean. Prioritized messages can be forwarded immediately as soon as they are received. Low priority messages can be sent only when the simulated time advances to their timestamp, because they can be preempted by high priority messages.

The message population in the network has a constant value specified by the user as an input parameter for the application. Other input parameters are represented by the dimension of the hypercube, the type and the mean of the distribution for the message service time in *LPs* and for the *LP* destination selection, the ration of low priority messages and prioritized messages.