

Curso 2010/11
CIENCIAS Y TECNOLOGÍAS/13
I.S.B.N.: 978-84-15287-34-6

IVÁN CASTILLA RODRÍGUEZ

**Explotación de los sistemas multi-núcleo para
la simulación paralela de eventos discretos con Java.
Ejemplo de aplicación en un modelo orientado
al proceso de un centro hospitalario**

Directora
ROSA MARÍA AGUILAR CHINEA



SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

A Arelis...

A mis padres...

Agradecimientos

Esta tesis está escrita en inglés, lengua franca de la ciencia. Aunque agradecer tiene algo de ciencia, manejar con sutileza esta ecuación de sincero reconocimiento requiere un lenguaje en el que muestre mayor destreza.

En primer lugar, querría agradecer a la Dra. Dña. Rosa M^a Aguilar China su dirección, guía y apoyo durante la realización de este trabajo. Su ambición e impulso me han empujado a cotas que, de haber dejado en mis conservadoras manos, nunca me hubiese planteado alcanzar. Congresos, estancias, publicaciones... son producto de su insistencia y su confianza incondicional en mi trabajo. Gracias Rosi.

Al Dr. D. Lorenzo Moreno Ruíz le debo mi incorporación al mundo de la investigación en el Departamento de Ingeniería de Sistemas y Automática y Arquitectura y Tecnología de Computadores de la ULL (todavía Grupo de Computadoras y Control por aquellos días). Sin su tutela y sus consejos durante mis primeros años, esta tesis no hubiese sido posible.

Quiero agradecer muy especialmente la colaboración y ayuda desinteresada del Dr. D. Félix García López. Durante estos últimos meses he gastado completamente la expresión “gracias” con él, así que sólo me queda dedicarle este pequeño reconocimiento.

D. Roberto Muñoz González y D. Antonio Yeray Callero de León han sido mis “segundos de a bordo” en dos etapas sucesivas (entiéndase “segundos” como una cuestión de antigüedad, nunca de importancia) y han sido el mejor apoyo para mi trabajo y esa oreja fresquita para escuchar mis interminables divagaciones. A Rober, como ya era amigo de antes, tengo que agradecerle que, pese a mi cháchara, siga siéndolo todavía; a Yeray, que se haya convertido en uno.

A D. Carlos Martín Galán le debo, literalmente, la idea de esta tesis. Su trabajo es la base sin la que todo esto no sería posible.

Throughout this research I have had the invaluable opportunity to visit two foreign research groups: the Department of Management Science at Lancaster University (UK), and the Communication Systems Group of the University of

Bonn (Germany).

In Lancaster, I would like to thank Dr. Murat Günal (currently at the Turkish Naval Academy - Turkey) and Dr. Mike Pidd for their amazing hospitality. I learnt a lot about simulations and hospitals with them, but especially about good research techniques. I would also like to thank Murat for the only nice meal I had in England (of course, it was Turkish food), and Mike for showing me the Lake District.

In Bonn, I would like to especially thank Dr. Patrick Peschlow and Dr. Peter Martini. I would like to thank Patrick for the long discussions about parallelism and his smart ideas and contributions to my research. Moreover, Patrick took care of every little detail and made my visit the most pleasant experience ever. Thanks to him, I hardly noticed my complete ignorance of the German language. Of course, I have to include my appreciation to all of Patrick's colleagues, who took me with them to *Mensa* and treated me like family: Jakob Bieling, Simon Schwarzer, Matthias Frank, Alexander Willner, Lukas Pustina. . . *Vielen Dank!*

Volviendo al mundo hispano, tengo que dar las gracias al personal del Hospital Universitario Nuestra Señora de Candelaria, en especial a D. Carlos Bermúdez Pérez y D. Juan Antonio Gil Martín.

Del otro hospital tinerfeño, el Hospital Universitario de Canarias, debo darle las gracias a Dña. Gregoria Torres González, por la información sobre la organización hospitalaria y a Dña. Concepción Rodríguez González, por la colaboración para la validación del modelo conceptual del hospital. En la citada validación colaboró también el Dr. D. Alfonso Castilla García, del Servicio Canario de Salud. Como, casualmente, Dña. Concepción y D. Alfonso son también mis padres, no puedo limitarme a agradecer el apoyo moral de la familia, que lo ha habido en ingentes cantidades (gracias también a mis hermanos Ana y Pablo), sino que debe resaltar la importancia de su aportación en la finalización de una parte tan compleja como es el modelo del hospital. Gracias, de corazón.

Debo agradecer al Dr. Juan Albino Méndez Pérez (Alexis) por haber sido un gran consejero y un mejor AMIGO durante todo este tiempo.

Tengo que destacar a D. Jonatán Felipe García, D. Pedro Antonio Toledo Del-

gado y al Dr. D. Jesús Torres Jorge porque llevamos muchos años (y más de un verano) en esto. Su compañía, su ayuda y su amistad han aparecido más veces de las que puedo recordar. A Jesús, especialmente, le debo esta tesis, o al menos la plantilla latex con la que está hecha.

Quiero agradecer también a los chicos (y chica) del laboratorio, compañeros sempiternos de desayunos y/o mesa y soporte de mi ácido humor, más y más corrosivo según se acercaba el final de este trabajo. No me olvido de D. Rafael Arnay del Arco, D. Jesús Javier Espelosín Ortega, Dña. Ángela Hernández López, D. Néstor Morales Hernández, D. Daniel Perea Ström y el Dr. D. Santiago Torres Álvarez.

Por supuesto, quiero dar las gracias al resto de compañeros del departamento que en uno u otro momento me han ayudado, como el Dr. D. José Ignacio Estévez Damas, el Dr. D. José Francisco Sigut Saavedra o el Dr. D. Evelio José González González, así como al resto de compañeros: Dr. D. Leopoldo Acosta Sánchez, Dr. D. José Demetrio Piñeiro Vera, Dr. D. José Luis Sánchez de la Rosa, Dr. D. Alberto Francisco Hamilton Castro, D. Juan Julián Merino Rubio, Dr. D. Graciliano Nicolás Marichal Plasencia, Dr. D. Roberto Luis Marichal Plasencia, Dra. Dña. Marta Sigut Saavedra, Dra. Dña. Carina Soledad González González, Dra. Dña. Silvia Alayón Miranda, Dra. Dña. Vanesa Muñoz Cruz, Dr. D. Jonay Tomás Toledo Carrillo, D. Germán Carlos González Rodríguez, D. Hector Javier Rebozo Morales, D. Sid Ahmed Ould Sidha, D. Ginés Coll Barbuzano y D. Eladio Hernández Díaz.

No me olvido de los bravos muchachos que han colaborado para hacer crecer a SIGHOS: Dña. Yurena García-Hevia Mendizábal, Dña. Alicia de Fuentes Arteaga, D. Rayco Hernández de León, D. Rayco Díaz Batista y D. José Carlos Díaz Rodríguez.

Pese a la extensión de estos agradecimientos, no puedo dejarme atrás a mis amigos Jorge Paiz Sosa, Abraham Medina Herrera, Israel Pérez Pérez y Leticia Quintana Expósito, que han colaborado a rebajar el nivel de stress hasta límites tolerables. También quiero agradecer el apoyo incondicional de Juan Ramón González González, Isabel Sánchez Berriel y Virginia Gutiérrez Rodríguez. Los

“dones” les sobran a todos ellos.

Probablemente he dejado para el final a quien quizás es la persona más importante de este proceso. Arelis ha convivido conmigo la mayor parte del tiempo que he estado trabajando en esta tesis y ha soportado mal humor, depresiones, nervios y pesimismo exacerbado. . . pero nunca me ha dejado solo ni ha dejado de animarme. Si alguien sabe cuánto ha costado esto, es ella.

Finalmente, debo agradecer al Ministerio de Educación por haber financiado mi trabajo estos últimos cuatro años con una beca de Formación de Profesorado Universitario (FPU) nº AP2005-2506.

Contents

Introduction	xxix
Overview	xxix
Research Objectives	xxxiii
Structure of the Thesis	xxxv
Chapter 1. Theoretical Foundations for Parallel Discrete Event Simulation	1
1.1. Discrete-Event System Simulation	2
1.2. DES World Views	4
1.3. Parallel Discrete Event Simulation	11
Chapter 2. Process-Oriented Implementation	23
2.1. Basics of Implementing a Process-Oriented Simulator	24
2.2. Direct (Threaded) Translation	25
2.3. Coroutines	26
2.4. Stack Swapping	27
2.5. Continuations	28
2.6. Stack Reconstruction	31
2.7. Converting Process Interaction into Event Scheduling	34
2.8. Java for Process-Oriented Simulation	35
Chapter 3. SIGHOS: a Process-Oriented Simulator	39
3.1. Basic Definitions of Business Processes	40

3.2. Workflow Patterns	41
3.3. SIGHOS: A Process-Oriented Simulation Tool	44
3.4. Inside SIGHOS: an Event Heart	50
3.5. Inside SIGHOS: Managing Workflows	57
3.6. Inside SIGHOS:... Why It Matters?	67
Chapter 4. Parallel SIGHOS	71
4.1. From Sequential to Parallel	72
4.2. Limits to Parallelism: Resource Contention	75
4.3. Resource Contention in SIGHOS	77
4.4. A Test Benchmark for SIGHOS	87
4.5. A Performance Analysis of the Sequential SIGHOS	89
4.6. Using External Event Executors	93
4.7. Integrating the Pool in the Simulation Tool	95
4.8. Exploiting Event Locality	98
4.9. Block Dispatching	98
4.10. A Hybrid Event Manager - Executor	101
4.11. Going Beyond Limits: like-3-Phase Approach	105
4.12. Some Final Notes about the Implementation	116
4.13. Putting It All Together: Hybrid EME like-3-Phase Approach	123
4.14. Summary: Comparing the Different Approaches	125
Chapter 5. Case Study: A Model for Hospital Management	131
5.1. Overview of a Hospital	131
5.2. Hospital Performance: Why Modelling and Simulation?	133
5.3. Background	134
5.4. Review of the Literature on Modelling a Whole Hospital	141
5.5. A Modular Model for a Whole Hospital	146
5.6. Conceptual Model	147
5.7. Computational Model	155
5.8. Results	178

5.9. Conclusions	184
Conclusions	187
Contributions	189
Further Research	195
Apéndice A. Resumen	199
A.1. Planteamiento del Problema	199
A.2. Objetivos de la Tesis	202
A.3. Resultados y Contribuciones	203
A.4. Conclusiones	213
A.5. Líneas Abiertas	215
Appendix B. PSIGHOS: User Interface	219
Appendix C. Other Java DES Tools	251
C.1. JSIM	252
C.2. Simkit	253
C.3. SSJ	253
C.4. DESMO-J	254
C.5. JAPROSIM	256
Appendix D. Other Java PDES Tools	259
D.1. JUST: Java Ubiquitous Simulation Tools	259
D.2. SPADES/Java	259
D.3. D-SOL	260
D.4. CSA&S/PV: Complex Systems Analysis & Simulation - Parallel Version	261
D.5. Summary of Java PDES Tools	262
Appendix E. Workflow Patterns	265

Appendix F. Computational Concurrency	279
F.1. Awareness Level in the Interaction Between Processes	279
F.2. Competition between Processes for Resources	280
F.3. Cooperation between Processes by Sharing	281
F.4. Control Mechanisms	282
Bibliography	287

List of Tables

2.1. Features of a process-oriented simulation	24
3.1. Support of control flow patterns	65
4.1. Main goals of the intended solution	72
4.2. Simple example of the resource booking phase	86
4.3. Advanced example of the resource booking phase	87
4.4. Test platform	89
4.5. Sequential parameters	90
4.6. 10,000 Vs 100,000 iterations ($\alpha = 8$)	90
4.7. Three consecutive RoleOn events with the same timestamp	106
4.8. Combination of parallel optimisations	126
4.9. Parallel parameters	127
4.10. Final comparison of performance for $\alpha = 4$	127
4.11. Final comparison of performance for $\alpha = 8$	128
4.12. Speedup for H3P	129
5.1. Preliminary schedule of operating theatres for GS	139
5.2. Real usage of OTs: Elective surgical procedures in the morning shift 2004-2005	140
5.3. Main classes of the hospital model	157
5.4. Simulation components for central services	160

5.5. Parameters for central services	161
5.6. Simulation components for central laboratories	163
5.7. Parameters for central laboratories	167
5.8. Simulation components for medical departments	168
5.9. Parameters for medical departments	171
5.10. Simulation components shared by surgical departments	173
5.11. Parameters for shared surgical departments resources	173
5.12. Simulation components for surgical departments	174
5.13. Parameters for surgical departments	176
5.14. Resources of the HUC's central laboratories	179
5.15. Resource types and activities in the computational model	180
5.16. Number of RTs, As and AMs in each scenario	182
5.17. Speedup for different whole hospital model scenarios	183
B.1. Class TimeUnit	221
B.2. Class TimeStamp	221
B.3. Class TimeFunction	222
B.4. Class PeriodicCycle	222
B.5. Class TableCycle	223
B.6. Class Condition	223
B.7. Class Variable	223
B.8. Class Simulation	224
B.9. Class ResourceType	225
B.10. Class Resource	226
B.11. Class WorkGroup	228
B.12. Class TimeDrivenActivity	229
B.13. Class FlowDrivenActivity	230
B.14. Class ElementType	231
B.15. Class TimeDrivenGenerator	232
B.16. Class ElementCreator	232
B.17. Class Element	233

B.18. Class SingleFlow	234
B.19. Class ParallelFlow	234
B.20. Class ExclusiveChoiceFlow	235
B.21. Class MultiChoiceFlow	236
B.22. Class ProbabilitySelectionFlow	237
B.23. Class SimpleMergeFlow	238
B.24. Class MultiMergeFlow	238
B.25. Class SynchronizationFlow	239
B.26. Class DiscriminatorFlow	239
B.27. Class PartialJoin	240
B.28. Class InterleavedParallelRoutingFlow	240
B.29. Class DoWhileFlow	241
B.30. Class WhileDoFlow	242
B.31. Class ForLoopFlow	243
B.32. Class StaticPartialJoinMultipleInstancesFlow	244
B.33. Class StructuredPartialJoinFlow	245
B.34. Class StructuredSynchroMergeFlow	246
B.35. Class InterleavedRoutingFlow	246
B.36. Class StructuredDiscriminatorFlow	247
B.37. Class ThreadSplitFlow	248
B.38. Class ThreadMergeFlow	248
C.1. Summary JSIM	252
C.2. Summary Simkit	253
C.3. Summary SSJ	255
C.4. Summary DESMO-J	257
C.5. Summary JAPROSIM	258
D.1. Summary of Java PDES tools	262
E.1. Basic control flow patterns	266
E.2. Advanced branching and synchronisation patterns	267

List of Tables

E.3. Multiple instance patterns	271
E.4. State-based patterns	273
E.5. Cancellation and force completion patterns	275
E.6. Iteration patterns	276
E.7. Termination patterns	277
E.8. Trigger patterns	278

List of Figures

1.	Schematic structure of this thesis	xxxvi
1.1.	Events, activities and processes	5
1.2.	Event and activity orientations	7
1.3.	Three-phase approach and process orientation	9
1.4.	Logical Process Simulation	16
1.5.	Different PDES approaches	22
3.1.	Basic workflow terminology	41
3.2.	Resource availability	45
3.3.	Life cycle of an Element	49
3.4.	Life cycle of a Resource	55
3.5.	Basic workflow interfaces in SIGHOS	60
3.6.	Initializer workflows	61
3.7.	Finalizer workflows	62
3.8.	Structured workflows	63
4.1.	Basic schema of the Master-Slave approach	75
4.2.	Overlapping timetable entries	78
4.3.	Construction of Activity Managers	80
4.4.	Resource simultaneously available for roles belonging to different activity managers	82

4.5. Example of several elements requesting activities in different AMs	85
4.6. Comparison of influence of $ E $ and $ A $ for $\alpha = 8$ and $\beta = 1$	91
4.7. Effect of varying α for different problem types and $\beta = 1$	92
4.8. Effect of varying β for different problem types and $\alpha = 8$	92
4.9. Speedup using Java thread pool with $ E = 512$ and $\alpha = 8$	94
4.10. Speedup using Java thread pool with $ E = 512$ and $\alpha = 4$	95
4.11. Speedup using an ad hoc thread pool with $ E = 512$ and $\alpha = 8$	97
4.12. Basic schema of the optimised Master-Slave approach	99
4.13. Speedup exploiting event locality with $ E = 512$ and $\alpha = 8$	100
4.14. Speedup exploiting block dispatching with $ E = 512$ and $\alpha = 8$	101
4.15. Relaxed MS approach by using the hybrid EME	102
4.16. Speedup compared with $ E = A = 512$, $\alpha = 4$ and $\beta = 1$	103
4.17. Performance comparison with $ E = A = 512$, $\alpha = 4$ and $\beta = 1$ for different <i>grain-rest</i> configurations	104
4.18. Relationships among events	112
4.19. Speedup for the like-3-phase approach with $ E = 512$ and $\alpha = 8$	115
4.20. Comparison between the like-3-phase approach and the block dis- patching method with $ E = A = 512$, $\alpha = 8$ and $\beta = 1$	116
4.21. Comparison between using <code>java.util.concurrent.Semaphore</code> and a spinlock with a <code>java.util.concurrent.atomic.AtomicBoolean</code> with $ E = A = 512$, $\alpha = 4$ and $\beta = 1$	118
4.22. Comparison between extending <code>Thread</code> and implementing <code>Run- nable</code> with $ E = A = 512$, $\alpha = 4$ and $\beta = 1$	120
4.23. Performance of different implementations of barriers	122
4.24. Speedup comparison among a pure master EM, a hybrid EME, and a hybrid EME using a tournament barrier for synchronisation	123
4.25. Speedup for the hybrid EME like-3-phase approach with $ E =$ 512 and $\alpha = 8$	124
4.26. Comparison between the like-3-phase approach and its hybrid EME version with $ E = A = 512$, $\alpha = 8$ and $\beta = 1$	125

5.1. HUC's organisational chart	132
5.2. Basic diagram of the decision aiding tool. Source: (Moreno et al., 2000)	135
5.3. Basic schema for surgical patients at the HUNSC	137
5.4. IO schema of the simulation model	139
5.5. Comparing two scenarios with different OT configurations	141
5.6. Hospital Model. Source: (Günel and Pidd, 2008)	143
5.7. HADA. Source: (Castilla et al., 2008)	145
5.8. High-level conceptual model of a hospital	148
5.9. Flow for a medical patient	150
5.10. Flow for a surgical patient	151
5.11. Flow for ambulatory and short-stay surgical patients	152
5.12. Flow for an ordinary surgical patient	153
5.13. Diagnostic tests	154
5.14. Standard diagnostic test	155
5.15. Laboratory tests	156
5.16. Execution time for different whole hospital model scenarios	183
A.1. Aceleración de PSIGHOS sobre SIGHOS	210
A.2. Modelo conceptual del hospital	212
A.3. Tiempo de ejecución para diferentes escenarios de un modelo completo de un hospital	213

List of Code and Pseudocode Listings

2.1. Example for applying continuations	29
2.2. The previous example with continuations	30
2.3. Example for applying stack reconstruction	32
2.4. Method “one” from the previous listing with stack reconstruction	33
3.1. RequestActivity event	53
3.2. FinalizeActivity event	53
3.3. AvailableElement event	54
3.4. SIGHOS algorithm for event scheduling	56
4.1. Event Manager main loop	74
4.2. Basic definition of an ad-hoc Event Executor	96
4.3. First optimisation of an ad-hoc Event Executor	96
4.4. Modified main loop of an ad-hoc Event Executor	97
4.5. The schedule(e@ts) operation modified to exploit event locality .	98
4.6. Event Manager main loop using block dispatching	99
4.7. Main loop of an EE and the hybrid EME	102
4.8. RequestActivity event	107
4.9. FinalizeActivity event	107
4.10. AvailableElement event	108

4.11. RoleOn event	109
4.12. RoleOff event	109
4.13. AM.notifyAvailableResource method	110
4.14. AM event	112
4.15. Modified main loop of a like-3-phase Event Executor	113
4.16. Event Manager main loop using a like-3-phase approach	114
4.17. await() method with java.util.concurrent.Semaphore	117
4.18. await() method with java.util.concurrent.atomic.AtomicBoolean	118
4.19. Modified main loop of a hybrid EME like-3-phase Event Executor	124
5.1. A factory of simulation objects	159
5.2. Model of the USS service with SIGHOS	161
5.3. Model for the workflow of the USS service with SIGHOS	162
5.4. Model for central lab specialist nurses with SIGHOS	165
5.5. Model for a central lab laboratory test with SIGHOS	166
5.6. Model of a “waiting” activity with SIGHOS	170
5.7. Model for a flow-driven activity within SIGHOS	170
F.1. Use of Lock access methods	282
F.2. Comparison between Lock and RLock acquisition	283
F.3. Typical algorithm with a barrier	284

List of Acronyms

API	Application Programming Interface	160
BP	Business Process	187
BPR	Business Process Reengineering	42
BPS	Business Process Simulation	187
CZ	Conflict Zone	83
DES	Discrete-Event System Simulation	251
DESS	Differential equation system specification	260
DEVS	Discrete Event System Specification	260
EE	Event Executor	188
EM	Event Manager	190
EME	Event Manager Executor	191
FEL	Future Event List	190
GC	Garbage Collector	36
GVT	Global Virtual Time	20
HLA	High Level Architecture	21
IT	Information Technologies	40
JVM	Java Virtual Machine	26

LP	Logical Process	15
LVT	Local Virtual Time	15
OO	Object Orientation	36
PDES	Parallel Discrete Event Simulation	259
PDEVS	Parallel Discrete Event System Specification	14
TWLP	Time Warp Logical Process	19
VC	Virtual Clock	190
V&V	Verification and Validation	142
WfMC	Workflow Management Coalition	40

Hospital

A&E	Accident and Emergency	194
GP	General Practitioner	143
GS	General and Digestive Surgery	137
HES	Health Episode Statistics	194
ICU	Intensive Care Unit	195
IP	inpatient	133
MD	Medical Department	147
NHS	National Health System	134
OP	outpatient	133
OT	operating theatre	
PACU	Postanaesthesia Care Unit	195
PAS	Patient Administration System	194
SD	Surgical Department	147

Introduction

Overview

Nowadays, *information* constitutes one of the most important values of an organisation. As a result, Information Technologies (IT) have risen to the forefront of investment and research for both private companies and public institutions. IT are defined by the Information Technology Association of America (ITAA) as

« ... the study, design, development, implementation, support or management of computer-based information systems, particularly software applications and computer hardware. »

Though the information systems of complex organisations offer a more suitable scenario for benefiting from technology, at the same time, the implementation of an efficient and effective system requires further and more careful planning and design as the size of the organisation increases and the interrelations among its departments, employees and material resources become more intricate.

Business Process

Service organisations are typically structured based on the *division of labour principle*, that is, workers are grouped and specialised in certain tasks and roles and arranged in a hierarchical manager-supervised structure. Within this context,

three different types of *processes* can be defined: material, information and business (Medina-Mora et al., 1992).

Material and *information* processes represent different views of an organisation, and focus respectively on the physical products that the organisation handles, and on the information that is created, processed, managed and provided. The Workflow Management Coalition (WfMC, 1999) describes a Business Process (BP) as:

« A set of one or more linked procedures or activities that collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships. »

In contrast to material and information processes, BPs allow for a holistic approach to the organisation in terms of higher-level market-centred goals. Hence, the correct identification and modelling of an organisation's BPs represents a great opportunity to both properly exploit and improve the performance of IT in a business context.

Simulation as a Tool for Decision Making

Though the advantages of the division of labour are undeniable, over time, this way of working leads to a fragmentation of the business that negatively impacts on costs and on staff motivation. Furthermore, managers have to supervise more and more complicated processes due to an increase in the number and variety of tasks to be performed. This, along with the changes taking place in corporate settings, such as more demanding clients, the need for increased competitiveness and market innovation means that a rigid organisational structure is not the best option for today's companies, which must be organised around their processes.

One of the disciplines that has emerged to address this problem is *Business Process Reengineering (BPR)*. Muthu et al. (2006) expose that

« Reengineering is the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance such as cost, quality, service and speed. »

Reengineering, like any other decision-making process, whether in companies, industry or government, requires an in-depth understanding of the underlying problem. Control actions cannot be taken based only on assumptions. There is a pressing need for reliable analysis techniques that minimise the costs resulting from decision-making errors. One such technique is computer simulation. Hence, *Business Process Simulation (BPS)* has emerged as an important tool within BPR. As described by Wynn et al. (2008), BPS is intended to achieve two main goals:

1. To analyse a process's behaviour, by developing accurate simulation models;
2. To understand the effects of running that process by executing simulation experiments.

Of course, these goals are unaffordable if a suitable BPS is not available. A well-designed BPS requires:

- basic model building blocks, including entities, resources, activities and connectors;
- activity modelling constructs, such as split, join, branch and assemble;
- and advanced modelling functions, such as attributes, expressions and resource schedules.

Bosilj-Vuksic et al. (2007) show that Discrete-Event System Simulation (DES) is well-suited to coping with the aforementioned goals.

The Importance of Parallelism in Simulation

There are many circumstances in which the appearance of complex models is inevitable (Chwif et al., 2000). For example, BPS is a field that generally involves complex models due to the dimensions and existing interrelationships between systems.

Though no conclusive studies exist that characterise the relationship between complexity and computational cost, it seems logical to assume that the greater the model complexity, the greater the simulation's need for computational resources in terms of both CPU and memory. Moreover, the quality of a simulation study can be seriously affected by the efficiency of the simulation tool: correcting a modelling error after a two-minute simulation is completely different than having to wait several days to obtain the same result.

Parallel/Distributed simulation is a technology that enables a simulation program to be executed on a computer system comprising multiple processors interconnected by a communication network. The distinction between parallel and distributed simulation lies in the characteristics of the computer system on which the simulation is to be run. A parallel computer generally consists of several processors that are physically close and share memory and I/O devices. Hence, the *communication latency*, that is, the delay in transmitting a message from one processor to another, is relatively low. In contrast, distributed computers typically consist of several heterogeneous computers interconnected by means of standard communication protocols, such as Ethernet, thus resulting in higher communication latencies.

Though distributed simulations can achieve multiple goals, such as increasing fault tolerance, integrating simulation environments and handling a geographically distributed simulation, parallel simulations are more appropriate when the aim is to reduce a complex simulation's execution time.

Having mentioned that DES is well-suited to BPS, Parallel Discrete Event Simulation (PDES) (Fujimoto, 2000) is the logical option to try to exploit the parallelism in these problems.

Multi-Core Systems for Parallelism

Parallelism has been traditionally regarded as a pure research topic: a technique that is unaffordable for most organisations since it requires very expensive parallel computers or a complex and maintenance-intensive computer cluster. In recent years, the popularisation of multi-core computers for the home market has changed the previous scenario and opened the door to a wider selection of parallel software applications.

Multi-core computers, initially called *on-chip multiprocessing* or *single-chip multiprocessing*, consist of two or more processor cores placed on a single die. The multiple cores generally share some structures, such as second- or third-level cache or memory, and I/O buses. Nowadays, a typical desktop computer consists of no more than eight cores, and servers can feature 16, 32 and even more processors (if several tens of cores are involved, the term *many-core* computers is used). Hennessy and Patterson (2007) review the main concepts involving multi-core computers in their chapter on Multiprocessors and Thread-Level Parallelism.

Research Objectives

The overview makes it clear that BPS is a powerful tool and that parallelism is well-suited to improving the efficiency of a simulator. However, *parallel BPS* is a term that rarely appears in the literature. Parallel simulation and, more specifically, PDES is typically applied to simulate not only telecommunication networks and physical systems (biological, medical. . .), but also distributed multi-player gaming. Examples of PDES applications in the military area include battlefield simulations and emergency event training exercises.

To the best of our knowledge, only two previous contributions tried to improve the efficiency of a BPS by means of parallelism. Ferscha and Richter (1996) utilise Petri Nets to model BPs, which are subsequently simulated in a *massively parallel simulation engine*. Zarei (2001) also proposes the use of PDES to reduce the execution time of a BPS that is modelled by using control flow

graphs (CFGs). Both present a case study and apply domain decomposition (see Subsection 1.3.6).

In our opinion, these approaches have two main drawbacks. First, they are based on low-level or very schematic constructs (Petri Nets and CFGs respectively). While this is not counterproductive in and of itself, it does mean that a strong background in BP modelling is not enough to cope with their methodologies. Second, parallelism cannot be automatically exploited in their problems. Instead, a prior analysis and an explicit construction of the parallel simulation is required.

This thesis is intended to further this field by adhering to three main principles:

- *Generality*. This thesis does not focus on a specific problem or model within the BPS field. We are not interested in obtaining an extremely efficient parallel simulation for a specific organisation, but rather a *sufficiently good* parallel simulation algorithm that can be used to model and simulate any set of BPs.
- *Automation*. Parallelism must be exploited in a way that is transparent to the user, that is, without the user's intervention.
- *Efficiency*. Given a model, the parallel simulator must outperform a sequential simulation.

In conclusion, this thesis aims to explore the feasibility of a *generic* BPS that can *automatically* and *efficiently* exploit the parallelism of the model being simulated in a multi-core system.

Several sub-objectives can be derived from the main research goal:

1. Identify the characteristics that make DES and, more specifically, process-oriented simulation a suitable approach for modelling and simulating BPs.
2. Discuss the feasibility of automatically exploiting the parallelism in a BP model.

3. Establish the potential of a generic PDES to model and simulate organisations.
4. Explore techniques to reduce resource contention in a shared-memory parallel simulation.
5. Analyse techniques and algorithms that improve simulation performance.
6. Establish the suitability of Java as a language for implementing the proposed simulation.
7. Study the performance of this kind of parallel simulator in a multi-core computer.
8. Validate the usefulness of the approach by applying it to a real-world problem.

Structure of the Thesis

Figure 1 summarises the structure of this thesis. The remaining chapters are organised as follows:

Chapter 1 provides a theoretical background on DES and PDES. With respect to DES, the main concepts on the representation of time in a simulation and the different DES world views are reviewed. The main approaches to PDES are also analysed and organised into three main groups: application-level parallelism, simulation-level parallelism and model-level parallelism.

Process-interaction is one of the most widely used DES world views. *Chapter 2* presents an in-depth review of the different techniques utilised to create a simulator that implements said world view. Special emphasis is placed on the way Java can be used to solve the problems posed by this approach.

Chapter 3 establishes the basis for the need for simulation tools in organisational environments, and defines the most important terms pertaining to BPS and

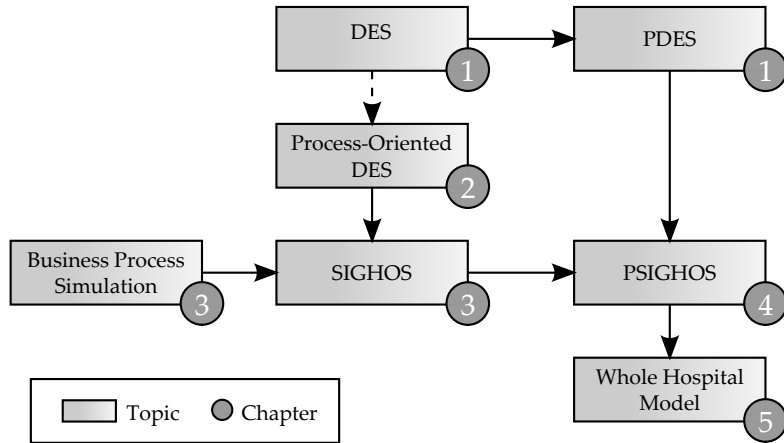


Figure 1. Schematic structure of this thesis

workflow modelling. Then, SIGHOS, a process-oriented DES library implemented in Java and intended for BPS, is described.

Chapter 4 studies the different approaches for automatically exploiting the parallelism in a DES tool like SIGHOS. The role of the resource contention problem is emphasised and several mechanisms for reducing its negative effect on the performance of the simulation are proposed. In an effort to maximise efficiency and performance, different techniques are applied and several modifications of the original algorithm are proposed. Every enhancement is validated with a practical implementation of the parallel simulator, called PSIGHOS.

Chapter 5 presents a case study to test the real-world utility and performance of PSIGHOS. The case study is based on research conducted by the Simulation Group from the Universidad de La Laguna, and describes the modelling and simulation of a whole hospital.

Theoretical Foundations for Parallel Discrete Event Simulation

According to systems theory, a *system* is a set of entities that interact to reach a specific common objective. Any “real” system, whether already existent or merely planned, is usually denoted as a *physical system*.

When a system is placed under study (Banks et al., 2000), not only do the system’s objects of interests (*entities*) have to be identified, but so do their properties or *attributes*, which can be used to distinguish among different entities of the same type, or to define their state over time.

Simulation is the imitation of a physical system’s behaviour over time, and can be carried out by developing a simulation *model*. A model represents the entities of the physical system and their interactions, either mathematically, logically or structurally. Generally, with the exception of very simple problems, computer-based simulation is the standard tool for performing simulations.

The *system state* is defined as the set of variables (henceforth *state variables*) required to completely describe the system at any simulated time. The description is *complete* whenever the objectives of the simulation study are covered. Consequently, developing a simulation model generally involves simplifications. It is unusual (and, generally, unaffordable) to make use of all the entities (and much less all the attributes) of the physical system in the model.

1.1. Discrete-Event System Simulation

Systems are often categorised as *continuous* and *discrete*. On the one hand, a *discrete system* involves changes of the state variables only at a discrete set of points in time. On the other hand, state variables change continuously over time in a *continuous system*. A system is rarely completely encompassed within one of these two categories, but this classification is still used for convenience.

Continuous and Discrete models can be defined in an analogous manner to systems. However, a discrete simulation model can be used to model continuous systems and vice versa.

This thesis focuses on *Discrete-Event System Simulation (DES)*. This kind of simulation mimics systems in which the state variables change only at a discrete set of non-regular time intervals. The instantaneous occurrences that may change the state of the system are termed *events*.

1.1.1. A Theoretical Framework for Representing Time in DES

The *time* concept is very important in DES and should be defined so as to avoid any kind of ambiguity. Loper (2002) defines in her thesis a theoretical framework to represent time in DES, divided into five dimensions: *Time, Clock, Time Flow, State Updates* and *Interactions*.

Time

Time can be seen from three different points of view (Fujimoto, 2000):

- *Physical time* is the time as it is expressed in the real or physical system being modelled.
- *Simulation time* is the simulator's representation of time.
- *Wallclock time* is the time the simulation execution lasts for.

Clock

Simulation time is used to order the events and is controlled by a *clock*. Such a clock holds the local notion of time during the simulation and assigns timestamps to every new event in order to define the instant at which that event will occur.

A simulation clock can be either physical or virtual. Any processor includes a hardware clock that can be employed as a simulation's *physical clock*. Since clocks belonging to different physical processors are not intended to be synchronised, a physical clock is useless in a distributed simulation unless there is a global wallclock time clock, keeping all the local clocks accurately synchronised. Continuous models are typically supported by physical clocks.

Virtual clocks were first introduced by Lamport (1978), when defining *logical time*. A virtual clock can assign to an event not only the current simulation time, but a time stamp that is greater than the current simulation time. An event with a timestamp greater than the current simulation time is scheduled for execution in the future. Discrete models are typically supported by virtual clocks.

Time Flow Mechanisms

Any simulation requires strategy for advancing time. Time flow mechanisms are also known as timing routines, event scheduling procedures or simulation executives. *Time-slicing* and the *next-event technique* (Pidd, 1998) are their most notable exponents.

Using a fixed time increment, which is the idea behind the time-slicing technique, the simulation model is examined and updated according to some pre-defined regular intervals. The size of the interval must be carefully chosen in a way that

- information is not lost;
- execution time is not degraded due to excessive clock updates with no changes in the system state.

The next-event technique is more suitable for models where consecutive events are irregularly distributed over time. The model is only examined and updated when a system state change is detected, that is, a new event occurs. Most DES tools employ this mechanism.

Nance (1971) presents a set of algorithms covering the whole spectrum from time-slicing to next-event. His conclusions state that the proper selection of the time flow algorithm has a dramatic influence on the simulation performance.

State Update

Any simulation defines a set of variables to represent the state of the physical system being modelled. When an event is executed, it modifies the value of one or more variables, thus updating the system state. State updates are consequently tied to the execution of events at a specific simulation time.

Interactions

Whereas state updates of an entity E_0 are changes due to the evolution of E_0 itself, *interactions* account for the influence of other entities $E_1, E_2 \dots E_n$ on E_0 . An *interaction* can be defined as an action of one simulation entity that explicitly affects another simulation entity. Contrary to state updates, which are persistent in time, an interaction is an immediate action that may affect the state of a persistent entity. Interactions are normally scheduled to be executed in the future.

1.2. DES World Views

DES involves several conceptual frameworks or world views. Balci (1988) described perfectly how to implement each one of these orientations, based on three basic components: event, activity and process.

- *Event* (e) is an instantaneous occurrence that modifies the state of an entity or the whole system.

- *Activity* (a) is a period of time that must pass before the state of an entity is modified. Generally, the start and end of an activity can be seen as events.
- *Process* (p) is a time-ordered sequence of events or activities.

Figure 1.1 represents the relationship among these three components, which are the basis of the world views described henceforth.

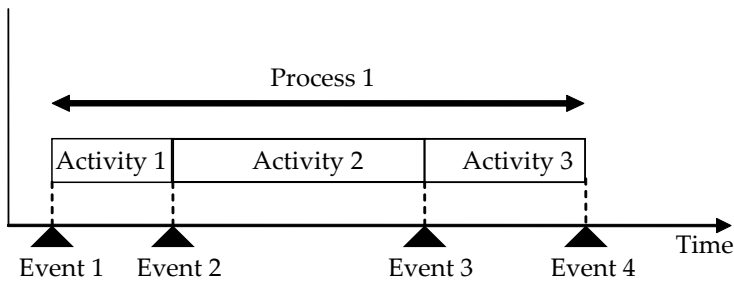


Figure 1.1. Events, activities and processes

1.2.1. Event Scheduling

Event oriented simulation is based on managing a Future Event List (FEL), containing all the events scheduled to occur at a future time. Events are allocated in the FEL in non-decreasing timestamp order. Therefore, let $e@t$ denote an event e scheduled to be executed at timestamp t .

The algorithm in Figure 1.2(a) shows how event orientation manages the FEL. Basically, the next-event technique, as formerly introduced in Subsection 1.1.1, is used. First, the simulation clock is advanced to t_1 , with $e_1@t_1$ being the front event from the FEL. Next, all events $e_2@t_2, e_3@t_3, \dots, e_i@t_i, t_2 = t_3 = \dots = t_i = t_1$ are moved to an execution queue. If the execution of an event produces new events, they are inserted in the FEL. When all events $e_k@t_k, t_k = t_1$ have been executed, the simulation clock is advanced again by taking the first event from the FEL. This process is repeated until the clock reaches the simulation end.

1.2.2. Activity Scanning

Using the activity orientation requires associating a series of *pre-conditions* to each activity. A time-slicing technique is usually applied to this kind of simulation. Every time the simulation clock advances, the associated pre-conditions are checked to determine if the activity can start, as seen in Figure 1.2(b).

Activity scanning is considered a simple, easy to understand and highly scalable approach. However, a computer execution of such an approach results in slow runtime because many useless scans have to be done each clock cycle. A logical and much more efficient evolution of this design is the three-phase approach, which will be introduced later this section.

1.2.3. Process Interaction

Process interaction is one of the most popular DES approaches, especially because of the use of *high-level blocks* to build models. The low-level simulation engine is responsible for handling the interactions among such blocks in a way that is transparent to the final user.

When this orientation is used, the life cycle of each entity is defined as a process. During a simulation, entities “flow” through the system, advancing as much as possible through their processes. Two different situations can make an entity stop:

- *Unconditional delay*: The entity is halted during a specified simulation time that is known or can be calculated at the instant the delay begins. Activities are generally considered unconditional delays.
- *Conditional delay*: The entity is halted until a certain condition is met, such as some system state variables reaching certain specified values. More often, conditions are based on the availability of certain resources required to perform an activity.

As mentioned earlier, process interaction has the advantage of offering the

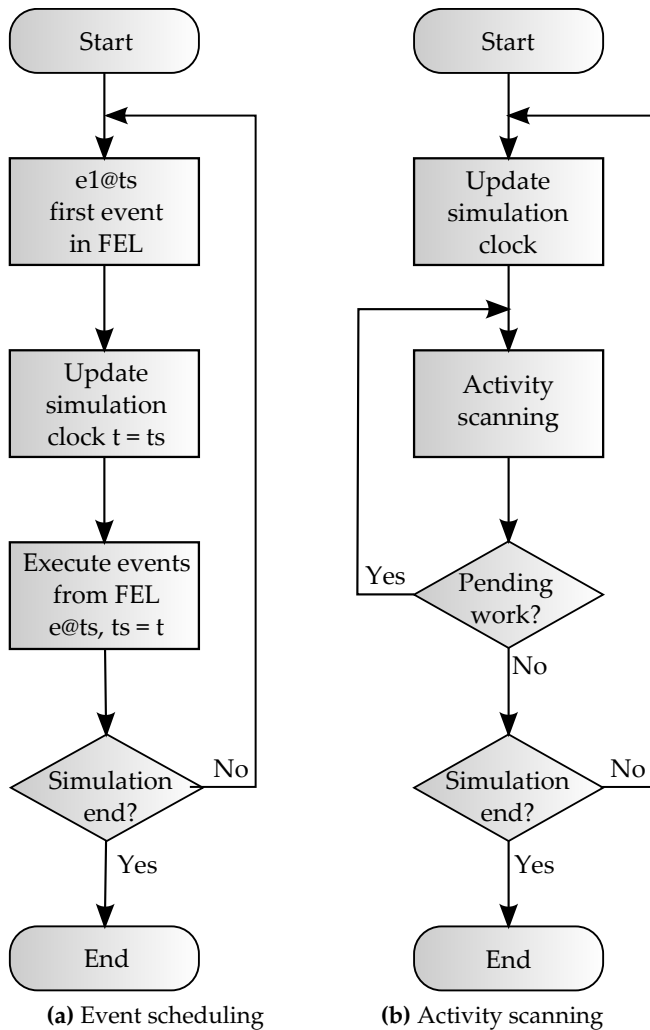


Figure 1.2. Event and activity orientations

modeller high-level structures for creating a model. At the same time, at a lower level, these processes are transformed into simpler structures (usually events). This approach, then, combines scalability and simplicity of design with efficient performance. Figure 1.3(a) shows a simplified scheme of this orientation. The

low level of the process interaction world view will be thoroughly examined in Chapter 2.

1.2.4. Three-Phase Approach

The three-phase approach combines characteristics from both event scheduling and activity scanning. Assuming as the starting point that events are activities lasting zero time units, activities/events¹ can be categorised into:

- *B activities/events*. Let $eb@t$ be the Bounded event eb , scheduled to be executed at simulation time t . Since the time instant when a B event is going to occur can be determined in advance, they can be scheduled in a future event list similarly to the event orientation.
- *C activities/events*. Let $ec@⟨cond⟩$ be the Conditional event ec , which will be executed if condition $cond$ is met.

The three-phase approach is drawn in Figure 1.3(b). *Three phases* are involved in the life cycle of this approach:

- *Phase A*: The simulation clock is advanced to t , as in the *event scheduling* approach.
- *Phase B*: B events $eb_i@ts_i$ whose timestamp $ts_i = t$ are executed.
- *Phase C*: C events are executed as in the *activity scanning* approach, according to a priority-based order. The execution of a C event can modify the system state, and thus some condition outcomes. Consequently, this phase has to be repeated until no more changes are detected.

¹Depending on the reference, the unit of work in the three-phase approach may be called an *activity* or an *event*.

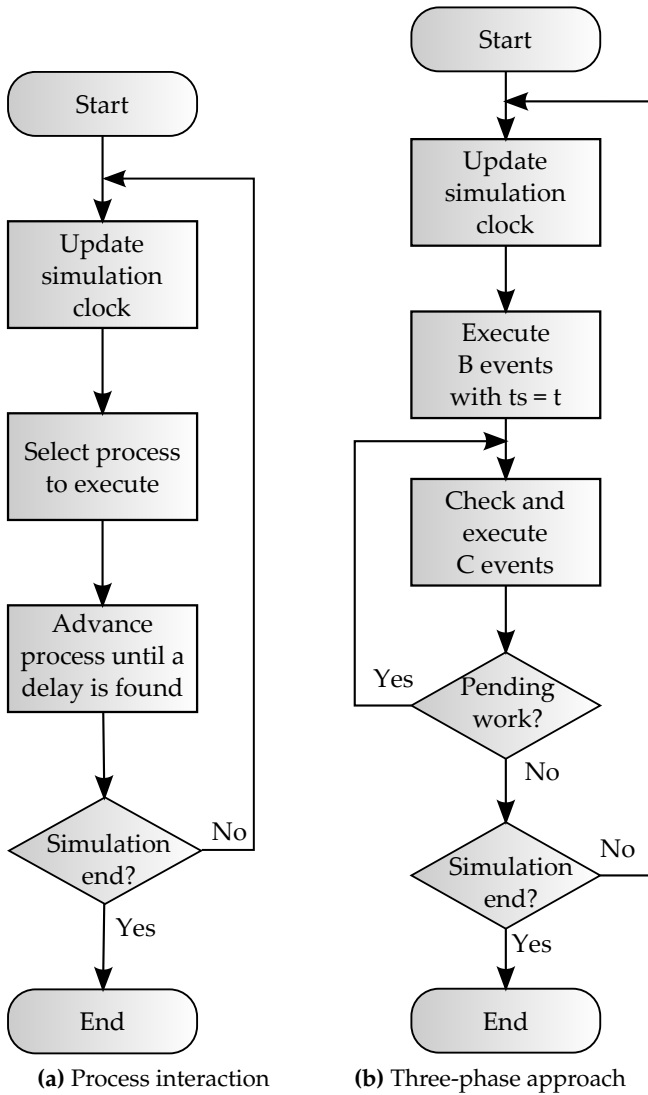


Figure 1.3. Three-phase approach and process orientation

1.2.5. Comparing the Four Approaches

Generally, simulation experts note the efficiency of event orientation when compared to other world views. Moreover, most modelling formalisms, such as Petri nets and Discrete Event System Specification (DEVS), are mainly geared toward analysing event-oriented models. Nevertheless, experts also agree on the difficulty of modelling using a pure event orientation. This difficulty arises mainly from the need to cover all the possible system evolutions within the event definition. This problem is further aggravated should the model have to be improved or extended.

The benefits and drawbacks of activity orientation pros and cons can be summarised in three main points: simple to model, easy to understand, but hardly efficient.

Process orientation is the most widely used worldview in both free and commercial simulation tools. Organisations find this approach well suited to their needs, specifically when modelling distribution and manufacturing processes. Not only is translating such systems into process-oriented simulation models very straightforward, but a process-oriented simulation also usually relies on an event-based simulation engine, thus improving performance.

The three-phase approach combines, according to its supporters, the simplicity of activity orientation and the efficiency of event orientation. Pidd (1998) adds an additional statement in support of this approach over process orientation: *event ordering*. Since the three-phase approach orders the execution of the events, conflicts when accessing shared resources are avoided. Therefore, the execution is deadlock free. Pidd's statement is based on the *sequential nature of the computer*:

« Any discrete simulation model that contains parallel and simultaneous activities that are to be run on a computer which is *essentially serial in operation* must have some strategy to avoid deadlock. »

This assertion, legitimate in 1998, is open to discussion nowadays. Multi-core computers are widely used today, even in the field of personal computers. Thus,

having at one's disposal 2, 4, 8 and even more processors to execute a simulation has become commonplace.

1.3. Parallel Discrete Event Simulation

Executing a simulation can take a long time. The complexity of the model or the length of time being simulated may result in the simulation lasting for hours, days or weeks, even in high-performance computers. Consequently, parallelism presents a good opportunity for reducing the execution time of a simulation.

Broadly speaking, techniques for exploiting simulation parallelism can be categorised into three main groups:

- *Application-level parallelism*. Techniques that do not modify the simulation core but which search for parallelism in the simulator's source code or simply distribute the simulator's main components. Very little or no knowledge of either the model or the simulation engine is required to implement these techniques. Among the techniques requiring no knowledge of the simulation at all, *automatic program parallelisation* stands out. Additionally, *distributed experimentation*, which can be applied to any stochastic simulation, and *dedicated execution*, which only requires modifying the interface among the main independent modules comprising the simulation tool, would fall into this category.
- *Simulation-level parallelism*. This category includes techniques that modify the simulation engine but that do not depend on the model being simulated. Dedicated execution would also fall into this category, although tangentially. The *distributed execution use of a centralised event list* is perhaps the only pure approach, since it modifies the simulation engine but not the model being used. Domain decomposition and cloning, roughly applied, would be included here, but the performance gain would not be remarkable.

- *Model-level parallelism*. It is not only the simulation engine that is modified when applying these techniques. Both the conceptual and computational models are defined so the parallelism present in the system being simulated is exposed. *Domain decomposition*, *hierarchical decomposition* and, to a lesser extent, *cloning* fall into this category. Since these techniques rely on specific knowledge of the system, impressive results can be achieved when correctly applied. However, by being so intertwined with the specific problem, they require a greater development effort when compared to the techniques discussed earlier. Moreover, most new problems must be developed completely from scratch.

The rest of this section is devoted to a more in-depth review of these techniques.

1.3.1. Automatic Code Parallelisation by Using a Specific Compiler

Automatic code (or program) parallelisation is a research area that has no direct bearing on simulation. *Dependence analysis* techniques detect regions of a source code that can be parallelised. Then, *straight-line code parallelisation*, *do loop transformations* and *parallelisation of recursive routines* are applied to automatically transform a sequential code into a parallel one. Banerjee et al. (1993) clearly present the main topics related to this area, whereas Hall et al. (1996) reference the Stanford University *SUIF* compiler, a tool specifically created for this purpose.

The generality of this approach is both its major drawback as well as its greatest advantage. On the one hand, high-level knowledge about the implicit parallelism in the simulation or the model cannot be exploited; on the other hand, the simulator developer does not need to explicitly design a parallel tool.

1.3.2. Replicated Trials or Distributed Experimentation

Stochastic simulation requires executing several replicas of the same simulation to obtain results that are *statistically significant*. Therefore, distributing the dif-

ferent replicas among a set of a processors is an extremely simple way to achieve speedup. Indeed, by having these replicas be completely independent, this approach ensures almost linear speedup with negligible programming effort. The limit to this linearity is imposed by communication costs, which should be insignificant when compared to the execution time of a simulation.

Memory constraints clearly limit distributed experimentation. If using a physically distributed environment, the entire simulated model must fit in each computer's physical memory. Furthermore, when trying to apply this technique in an multi-core computer, enough memory must be available to simultaneously allocate N replicas.

Heidelberger (1988) analyses the statistical properties of the estimators obtained by running parallel independent replications on a multiple processor computing system. In addition, Biles and Kleijnen (2005) place the emphasis on the use of the world-wide web in conducting large-scale simulation studies by distributing simulations created with generic software, such as Silk and Arena.

1.3.3. Dedicated Execution

A well designed simulator may be seen as a set of *loosely coupled* modules that interact one with another. A typical stochastic simulation consists of (at least):

- a random number generator;
- a simulation engine, which can be a simple event list manager;
- some kind of graphical interface;
- and some tools for compiling (and processing) statistics.

(Comfort, 1984) presented one of the first examples of dedicated execution. The performance of this technique strongly relies on the fine balance among the execution of the different modules. Unfortunately, event list management is quite often constrained by bottlenecking, as noted by Kesidis and Singh (1995).

1.3.4. Hierarchical Decomposition

When using hierarchical decomposition, events are decomposed into *sub-events* that may be concurrently executed. This approach requires a thorough knowledge of the system being modelled.

Hierarchical decomposition is strongly tied to DEVS theory (Zeigler et al., 2000) and was formerly introduced by Concepcion (1989). Parallel Discrete Event System Specification (PDEVS) appears as its most natural evolution (Chow et al., 1994). There is a large community of researchers focused on this sound formalism, which establishes a robust theoretical basis for analysing, modelling and simulating any system. However, such a strong theoretical basis makes simulation professionals very reluctant to adopt this approach in the private field, where adaptability, ease of design and immediacy are generally prioritised over robustness and even formal solution validation.

Despite the large amount of research being conducted in this field, such as that by Himmelspach et al. (2007), some authors are trying to “flatten” the hierarchical distribution in an attempt to gain more parallelism (Glinsky and Wainer, 2006).

1.3.5. Distributed Events (with Centralised Event List)

A different approach relies on maintaining a centralised future event list, as in traditional sequential DES, but profiting from the availability of several processing units to distribute the events. The simplest algorithm would look for the next available processor to execute the event with the lowest timestamp.

At first, this approach seems easy to implement and would avoid the need to completely redesign an already built simulator: a simple piece-by-piece substitution of the event engine would suffice. The use of a centralised event list is especially well suited to shared memory systems, where every processor can easily access the list. Unfortunately, it is precisely this global event list that bottlenecks this approach as the number of processors simultaneously accessing the list increases.

Obtaining good speedup by using this technique depends on the number of events that can “safely” be executed concurrently. Therefore, determining the “safety” of the events becomes the hardest task for the simulation to perform.

The use of a centralised event list will be revisited in subsequent chapters.

1.3.6. Domain Decomposition

The impact of the bottleneck caused by a centralised event list may be reduced by *decomposing* and distributing such a list among the available processors.

Domain decomposition treats the physical system under study as a set of sub-systems, each one sequentially simulated in a *Logical Process (LP)*. An LP manages its own list of *internal or local* events, and maintains a set of communication channels with other LPs to send and receive *external or remote* events. The importance of LPs is reflected in the name sometimes used to refer to this approach: *Logical process simulation* (Ferscha and Tripathi, 1996). The main components of this kind of simulation are shown in Figure 1.4 and enumerated below:

- The *Communication Channel* is the media used by the LPs to exchange messages.
- A *Communication Interface (CI)* is used to gain access to the communication channel.
- *Region (R)*. Let S be the set of system state variables; each LP_i can access a subset $S_i \subset S$, disjoint to state variables assigned to any other LP. This approach, therefore, does not allow for different LPs to share variables.
- The *Simulation Engine (SE)* executes the internal events of each LP, and creates new internal and external events, by advancing its *Local Virtual Time (LVT)*. An internal event ei is any event affecting only variables belonging to $S_i \subset S$; whereas external events ee affect variables belonging to $S_j \subset S (i \neq j)$.

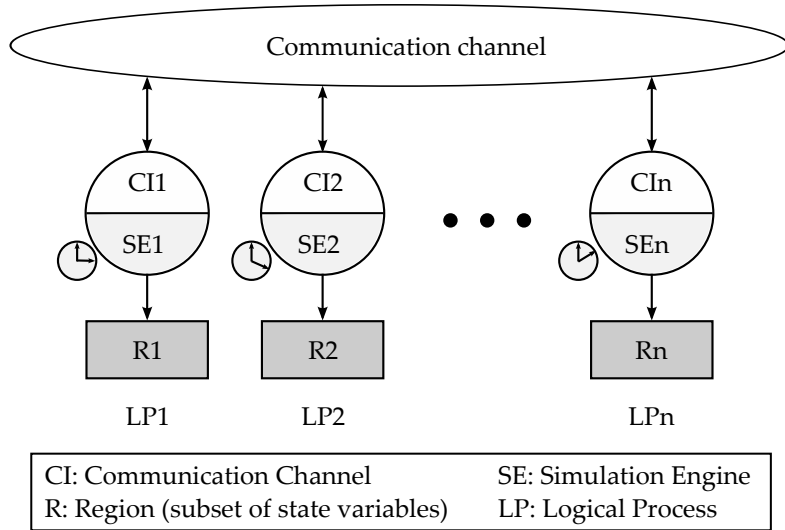


Figure 1.4. Logical Process simulation. Source: (Ferscha and Tripathi, 1996)

LPs must preserve the logical execution order of the events. With respect to local events, maintaining the order is as simple as executing events in timestamp order. However, since clocks from different LPs can have different values, it would be possible to receive a remote event $ee_k@t_k$ with $t_k < lvt$. A *causal error* appears if an already executed local event $ei_j@t_j$ with $t_j < t_k$ depended on ee_k . Preventing such causal errors from affecting the “correctness” of the simulation, which is known as the *synchronisation problem*, is the main research field for LP simulation experts. Here, “correctness” is not used to mean “validity,” as in Validation and Verification, but simply to denote a parallel simulation that produces the same results as the sequential one.

Consider the order in which the events are executed in a sequential simulation. Reproducing exactly the same global order in an LP simulation is the most trivial solution to the synchronisation problem, since it would ensure no causal errors at all. Unfortunately, no gain may be obtained by strictly adhering to a global sequential ordering. *Breaking* the order and executing events concurrently, thus

reducing the amount of synchronisations required, is the only way to improve upon sequential performance.

Generally, the synchronisation strategy falls into one of two categories: *conservative* or *optimistic*. The former is a *proactive* strategy, which tries to avoid causal errors; whereas the latter is a *reactive* strategy, which detects and fixes causal errors whenever they appear.

Conservative Synchronisation

As mentioned earlier, a correct simulation can be achieved if the global event ordering prevents the appearance of causal errors. A sufficient, but not necessary, condition to ensure correctness is that each LP adhere to the *local causality constraint*. As defined in Fujimoto (2000):

« A discrete-event simulation, consisting of (LPs) that interact exclusively by exchanging time stamped messages obeys the local causality constraint if and only if each LP processes events in nondecreasing timestamp order. »

This condition is not necessary since two events $ei_1@t_1$ and $ei_2@t_2$, with $t_1 < t_2$ and belonging to the same LP, may be executed in any order if there is no dependency relationship between ei_1 and ei_2 . Consequently, the “safety” of the next event $ei@t_i$ from the FEL has to be established before the event is actually executed. When adhering to the local causality constraint, ei will be “safe” if and only if it can be determined that no external event $ee@t_e$ with $t_e < t_i$ will be received.

Chandy and Misra (1979) and Bryant (1977) set the basis for conservative algorithms (and for Parallel Discrete Event Simulation (PDES) in general). They also highlighted the importance of using *null messages* ($0@t$) to avoid deadlocks among different LPs. A null message $0@t_nnull$ is an LP commitment that no event $ee@t$ with $t < t_nnull$ will be sent. Therefore, local events $ei@t$ with $t < t_nnull$ can be considered “safe.”

The definition of *lookahead* is another important milestone in the definition of conservative algorithms. An LP with local virtual time $lvt = t$ is said to have a lookahead L if said LP can accurately predict all the internal events to be generated until $t + L$. Combining null messages and lookahead allows an LP to move forward its *event safety horizon*. Computing the lookahead requires a detailed knowledge of the model being simulated, including characteristics of the communication pattern between LPs, minimum reaction time when a new event is received, conditional behaviour, precomputation of activity times, etc.

Conservative Time Window

The original conception of conservative algorithms is focused mainly on message passing schemes, used primarily in asynchronous distributed environments. Synchronism can be increased if suitable multiprocessor hardware is available and a time window W_i is used. A time window basically defines a set of events $e \in W_i$ that are *causally independent* from any event $e' \in W_j, j \neq i$. This idea was initially proposed by (Lubachevsky, 1988; Nicol, 1991) and is usually referred to as *Conservative Time Window* (CTW). The algorithm can be decomposed into two phases:

- *Phase 1: Window identification.* For each LP_i , an ordered series of events W_i is identified, such that for every event $e \in W_i$, e is causally independent of any $e' \in W_j, j \neq i$.
- *Phase 2: Event processing.* Each LP_i executes the events $e \in W_i$ sequentially and in chronological order.

A barrier is used to synchronise the two phases. Therefore, the benefits of this technique basically hinge on two main parameters: the efficiency of the synchronisation operations of the barrier, and the event structure of the model being simulated.

Optimistic Synchronisation

As conservative algorithms avoid the appearance of causal errors by adhering to the local causality constraint, optimistic algorithms allow breaking such constraints while offering a mechanism to recover (*roll back*) from any causal errors that do occur.

Despite the many algorithms that have been used, *Time Warp* (Jefferson, 1985) stands out as the most widespread and well known of these techniques, whose main concepts are commonly applied to establish the base for optimistic synchronisation. The Time Warp algorithm is generally split into the *local control* and the *global control* mechanisms.

The local control mechanism resides within each LP. A Time Warp Logical Process (TWLP) can be seen as a sequential discrete event simulator but for receiving events from different LPs, and not discarding events after processing them. Events are not discarded because a TWLP does not wait for synchronisation like conservative LPs, but optimistically executes all events in the FEL. As a result, a TWLP can receive an event from another TWLP with a timestamp preceding that of the last event processed. Such events are called *straggler events*. Upon receipt of a straggler event, the TWLP is aware that every event already processed whose timestamp is larger than that of the straggler event breaks the local causality constraint and is, consequently, incorrect. Therefore, the TWLP has to *roll back* or “undo” these events, and process them again after the straggler event has been processed. Such an “undo” operation poses two main problems: changes in state variables and events already sent to different TWLPs.

Although the local control mechanism ensures that the execution of the parallel simulation yields the same results as a sequential simulation, two additional problems remain which are solved by the global control mechanism: operations that cannot be rolled back, such as I/O; and the managing of the huge amount of memory required for roll back operations and which is no longer needed. Both problems can be solved if the simulation can safely determine that no straggler event will be received with a timestamp smaller than the simulation time T . This

lower bound is generally called *Global Virtual Time (GVT)*. All events and system state information prior to GVT can be safely deleted (or *fossil collected*). Computing the GVT is one of the most important problems in optimistic simulation.

Semi-automatic Domain Decomposition

Böszörmenyi and Stopper (1999) present a complementary approach for domain decomposition that tries to minimise causality conflicts at runtime by advancing the parallelisation to the earliest possible stage. The key idea is to enhance the model description with *hints* that describe the estimated workload and communication costs among simulation objects. Based on this information, a graph is created that is automatically partitioned, as expected in a classical domain decomposition.

The main advantage of this approach is its independence from the synchronisation strategy: both optimistic and conservative approaches are valid. Actually, the authors implement the latter for simplicity though, according to them, the former would be better suited to the system.

The main drawback is the difficulty in choosing the proper hints in advance.

Time Decomposition

Domain decomposition is classically defined as mapping *spatial* partitions of the simulated system into LPs. A different approach is to decompose the system *temporarily*, assigning different time intervals to different processors. In this case, combining the results from each time interval is the main challenge to using this approach.

Jones (1986) outlines this technique, whereas Chandy and Sherman (1989) present a framework and an algorithm to implement this idea. Heidelberger and Stone (1990); Lin and Lazowska (1991) cope with the problem of combining results. Fujimoto and Nicol (1994) also treat this kind of parallel simulation and note its limited scope: only problems that can be expressed as systems of recurrence relations can benefit from this technique; that is, systems whose temporal

behaviour does not depend on a previous state, and that can be recomposed with some simple operations. Kiesling (2006) recovers these ideas and introduces a progressive time-parallel simulation, a technique based on the successive refinement of solutions.

Final Remarks on Domain Decomposition

Domain decomposition is usually synonymous with PDES. This idea is reinforced by classical reference books such as (Fujimoto, 2000), which addresses this approach almost exclusively. Moreover, other techniques such as cloning (see next section), and IEEE standards such as *High Level Architecture (HLA)*, involving many important research and investor partners (like the US Army), are based on domain decomposition. It follows, therefore, that most of the scientific publications on PDES would fall into this research area.

Besides the aforementioned book by Fujimoto, Ferscha and Tripathi (1996) provide an in-depth review of the basic conservative and optimistic techniques, whereas (Perumalla, 2006) presents the most up-to-date survey on the state-of-the-art in domain decomposition.

1.3.7. Cloning

Cloning is a relatively recent PDES field based on the concurrent evaluation of several simulated futures (Hybinette and Fujimoto, 2001). The use of *decision points* distributed along the simulation timeline is a key feature of this approach. Decision points define highlighted instants when several copies of the currently running simulation are dynamically created, each one modelling a possible future outcome.

This technique mainly benefits from the fact that, independently of how many clones are created, computations performed before the cloning are required to be executed only once. Furthermore, the fact that it is based on domain decomposition means that only those parts (LPs) that actually change have to be cloned instead of the entire simulation.

Cloning, as introduced by (Hybinette and Fujimoto, 2001), is built atop LP simulation though, conceptually, it could be applied to any other parallel or even sequential approach. The use of LPs makes possible the incremental cloning of the simulation: only when an LP is affected by the decision is it actually cloned. A careful study of the moment when the LPs are cloned can lead to better results (Hybinette, 2004). Furthermore, clones that are detected as being equal can later be merged (Agarwal and Hybinette, 2005), thus avoiding unnecessary computation.

The main drawback of this approach, according to its authors, is that benefits from using cloning in stochastic simulations remain unclear, since computations with random numbers would differ very quickly.

1.3.8. Summary

Figure 1.5 comprehensively shows all the alternatives to PDES presented above. The different approaches are tentatively located along the X axis based on two opposing principles: on the one hand, approaches closer to the right require a deeper knowledge of the model and the simulation engine, and are thus more specific; on the other hand, approaches closer to the left are more generic, thus allowing for a more automated application. Certainly, the location of the alternatives is intended to be tentative, since hybrid approaches are possible and different variations of the pure alternatives are allowed.

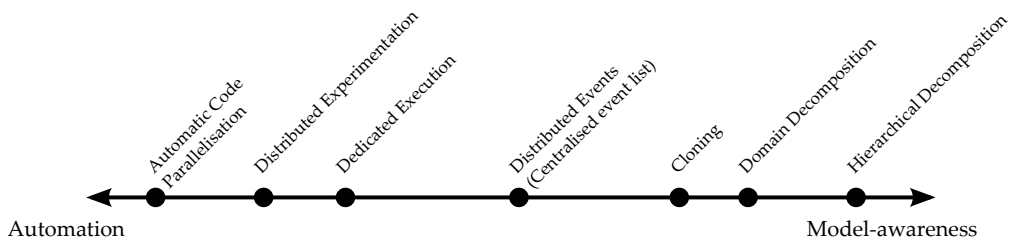


Figure 1.5. Different PDES approaches

Chapter 2

Process-Oriented Implementation

Process interaction is perhaps the most popular approach for implementing discrete event simulators. This popularity arises from several factors (Cassel, 2000):

- It is intuitive.
- Most popular simulation languages directly or indirectly support its use, from SIMULA (Dahl and Nygaard, 1966), the foundational simulation language, to SIMAN (Pedgen et al., 1995), heart of Arena, one of the most used simulation tools around the world.
- Processes seem to be the obvious way to link object-orientation and DES.

However, when dealing with parallel simulation, event scheduling is the preferred option. This statement may sound counterintuitive, since process orientation involves many processes *concurrently* interacting with one another. Therefore, a parallel approach should be suitable to implement such a world view. The rest of this section will try to shed light on this apparent contradiction by outlining the main techniques employed to implement the process interaction worldview in both sequential and parallel computers.

2.1. Basics of Implementing a Process-Oriented Simulator

According to (Perumalla and Fujimoto, 1998), a *pure* process-oriented simulator must offer the features shown in Table 2.1.

Table 2.1. Features of a process-oriented simulation

F1	Processes can declare and use local variables
F2	Process calls can be nested
F3	Processes can be recursive and re-entrant
F4	Primitives to advance simulation time can be invoked in any process
F5	Primitives to advance simulation time can be invoked wherever a conditional, looping or other statements can appear

Features 1-3 are covered by any high-level language supporting procedural programming styles. However, features 4-5 are simulation-specific and require further effort to be properly implemented. Certainly, depending on the application field, not all of these features are required, and the design of the simulator may be simplified.

Frequently, object-orientation is stated as being ideally suited for implementing process interaction (Jacobson, 1992) since an object's state is defined by a combination of variables, and its behaviour by a set of methods. The fundamental data structures in object-oriented programming are the different object classes. Therefore, to describe a process-based system, each of its elements must be modelled using an object that is an instance of one of the possible classes that exist in the system. Each particular instance of an object has its own state, even if the generic definition of its behaviour is set by the class to which it belongs.

An object-orientated language, such as Java or C++ can easily cope with features 1-3. However, features 4 and 5 involve the definition of an operation that may be invoked from the process body to suspend the current process during a conditional or unconditional delay. This operation has different names depending on the simulation tool: *suspend*, *hold*, *wait*... In spite of the different names,

using such an operation always implies that:

1. the simulation engine is able to halt the current process, thus allowing for a different process to be executed;
2. the state of the current process must be stored in order to be subsequently resumed.

The second implication involves storing not only local variables, but the exact execution point where the process was halted.

The rest of this section will thoroughly examine the different approaches involved in the efficient implementation of a suspend/resume scheme and highlight the details associated with a possible Java implementation.

2.2. Direct (Threaded) Translation

Threads are a convenient tool for circumventing the concurrent nature of process interaction. Threads can communicate and synchronise with each other, and generally include suspend/resume primitives. Therefore, each (simulation) process could be directly mapped onto a thread, and the suspend/resume primitives used to mimic the progression of the simulation timeline.

The main advantage of using threads is that the simulation engine becomes a simple thread scheduler and very little additional programming effort is required to capture the semantics of the process interaction. However, an extensive use of threads on a sequential computer is extremely inefficient, since thread creation and maintenance consumes a considerable amount of both CPU and memory resources.

The *recycling* of threads appears to offer a suitable alternative by limiting the cost of using direct thread mapping. A simulation mainly consisting of short-term processes (with respect to the entire simulation timeline) would derive some benefit from recycling threads mapped to already finished processes. However, the positive effect of this solution is bounded by the total amount of si-

multaneous processes in the simulation. A simulation of several thousand entities would rarely profit from this approach, even in a multi-core computer, which nowadays typically includes no more than 8 processors.

An additional problem, which only appears when Java is used, is that the *Thread.stop()*, *Thread.suspend()* and *Thread.resume()* methods are deprecated, that is, their use is discouraged by Sun because they can lead to deadlock-prone code. Consequently, extra hardcoding is required to re-implement such methods. Fortunately, the last versions of the Java Virtual Machine (JVM) include semaphores, locks and a variety of tools that make this task easier.

Mascarenhas and Rego (1996) apply this idea to create a distributed architecture that supports thread migration. Examples of sequential DES tools using this approach include Silk (Healy and Kilgore, 1997) and JAPROSIM (Abdelhabib and Brahim, 2008).

2.3. Coroutines

The efficiency problems of the direct translation make the adopting this approach impractical. Apart from being a convenient abstraction for process interaction, threads are not well suited to extensive use on a sequential, or even a standard multi-core, computer. A different approach is, thus, desirable.

As seen in Section 2.1, mimicking the process interaction approach on a sequential computer requires a mechanism for *stopping* a process whenever a conditional or unconditional delay arises; and for being able to restore the process state later, by *resuming the execution* at exactly the same point where it was previously interrupted.

Actually, the very first object oriented simulation language, SIMULA (Dahl and Nygaard, 1966), included a solution for this problem: the use of *coroutines*. According to Conway (1963), a *coroutine* can be defined as:

« ... an autonomous program that communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines

are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program. »

In other words, coroutines generalise subroutines to allow multiple entry points to suspend and resume execution at certain locations. Unfortunately, most general purpose languages such as C or Java implement subroutines using a stack-based approach (Koopman, 1989), which prevents the inclusion of coroutine primitives.

Certainly, coroutines can be mapped onto threads. Precisely, Helsgaun (2004) makes a thread-based implementation of coroutines in Java to support a process oriented simulation. Helsgaun proposes using a wait-notify scheme to suspend/resume such threads. Coroutines/threads already finished can be used again, thus saving some resources and reducing thread creation overhead. However, none of the disadvantages of this approach, as introduced in the previous section, are solved with Helsgaun's solution.

A different path is taken by (Weatherly and Page, 2004), who provide support for coroutines by modifying the IBM Jikes Reference Virtual Machine (The Jikes Team, 2010). Since the stack-based approach seems to be the problem, they replace the normal call stack by a *cactus stack* (Sardesai et al., 1998), which can easily deal with operations to suspend and later resume coroutines. Although their proposal appears suitable, their paper includes only very preliminary results.

2.4. Stack Swapping

Using high-level structures to introduce coroutine primitives in general purpose languages appears as a straightforward, but inefficient, mechanism. The use of low-level structures, then, is the next step.

As noted earlier, most general purpose languages use a *stack* for code optimisation. The stack is used to store a method's current state, thus allowing for subsequent resumption at the same point where it was interrupted to invoke

another method. Considering methods as simulation processes, and directly manipulating the stack would be a suitable approach for performing coroutine-like behaviour.

(Wiedemann, 2008) introduces assembler code that manually implements a stack swapping method. His approach allows for quickly switching context but requires storing stack pointer addresses for every process, thus buying performance with memory. The direct use of assembler code leads to several drawbacks, the most important being the strong dependency on the underlying OS/-hardware on which the simulator is to be executed. Furthermore, recurrent or re-entrant processes seems not to have been taken into account.

A similar approach is followed by (Jacobs and Verbraeck, 2004), who use the Java “assembler” code (*bytecode*) instead. Java does not compile into platform-specific machine code, but utilises an intermediate representation called *bytecode*. This intermediate code is *interpreted* by the JVM, which translates it into the platform-specific code. Being at a higher level, Java does not provide any language constructs to access the stack of a thread. Jacobs and Verbraeck (2004) dodge this problem by implementing a second Java interpreter on top of the *real* JVM. This new layer recodes the low-level bytecode operations and the structures used to interpret the code itself, but also adds primitives to access the *interpreted* stack. Hence, the *process* code is interpreted in the higher level layer, and the rest of the simulation code is executed as *normal* Java code. A simulation based on this mechanism is, according to the authors, around 6 times slower than an equivalent event scheduling simulation when there are more than 1000 simultaneously created entities. The time penalty is even higher when creating fewer entities.

2.5. Continuations

Continuations (Appel, 1992) have been also used to implement process interaction. This concept comes from compilers theory and, more specifically, from the

field of denotational semantics (Tennent, 1976). Programming with continuations implies explicitly telling each method about its successor (continuation). Let us consider, for example, the pseudocode in Listing 2.1.

Listing 2.1. Example for applying continuations

```
method f(x) {
    statement1;
    statement2;
    ...
    return r;
}

main {
    ...
    y = f(x);
    mainStatement1;
    mainStatement2;
    ...
}
```

Let k be a function enclosing the statements that take place immediately after the call $y = f(x)$. Listing 2.2 shows the result of applying the transformation.

Hence, k is the continuation of f . By reproducing the same transformation for the entire program, none of the methods would return except upon program termination. Consequently, the use of continuations obviates the program's need for a call stack.

One of the main difficulties with continuations is how to keep the context safe for local variables. Booth and Bruce (1997) solve this issue by ensuring that all local variables are constant (both values or references), and by defining a *closure environment* containing the references for external variables. Their simulator, named APOSTLE, implements an interpreter that translates a simulation expressed as a pure process-oriented language into C++ code that makes use of continuations. Unfortunately, once again, the results of applying such a solution are vaguely presented.

Listing 2.2. The previous example with continuations

```
method f(x, k) {
    statement1;
    statement2;
    ...
    k(r);
}

method k(r) {
    mainStatement1;
    mainStatement2;
    ...
}

main {
    ...
    y = f(x, k);
}
```

The authors say that

« We found that an interpreter for APOSTLE based on this technique was about as fast as our first generation compiler. »

By their “first generation compiler,” the authors mean a simulator that translates the process-oriented world-view into an event-scheduling approach, as will be explained later. They also claim that, by using continuations, they can take advantage of several optimisations that could be applied when transforming the code, such as merging two consecutive delays into a single one.

A different approach is that of Kunert (2008), whose efforts focus on the use of Java to develop an optimistic PDES. As the author says,

« Java does not provide a direct support for continuations. Moreover, it is impossible to implement generic continuations using pure

Java when one design goal is to keep the end-user unaware of the continuation implementation. »

Consequently, the only solution is to rewrite Java bytecode by using a specific continuation framework, such as (The Apache Software Foundation, 2010) and (The RIFE team, 2010).

2.6. Stack Reconstruction

Continuations try to completely avoid the use of the stack. In contrast, *stack reconstruction* (Perumalla and Fujimoto, 1998) involves not only using the stack, but adding further runtime information so that the stack can be reconstructed to the same state it was in when the process was suspended.

This technique consists of two main steps:

1. identifying (and labelling) all the lines of a procedure at which a process can be suspended;
2. and using a switch and a set of “goto” statements at the beginning of the process to select the proper part of the code to jump to.

Of the features introduced in Table 2.1, almost all of them are covered by this approach. The example in Listing 2.3 includes features 2, 4 and 5. It remains unclear how recursion (feature 3) is supported. However, the problems of local variables and procedure parameters (feature 1) are solved by creating a memory buffer or *frame*. Variables in the frame can be accessed by using indirect references, whereas the frame is kept together with the correct jump label.

Listing 2.4 illustrates how this technique can be applied. A variable “jumpIndex” would be required per process

Stack reconstruction is intended to overcome the classical difficulties that appear when applying process orientation in an optimistic PDES. Since the state information is stored in an abstract structure and not in the native stack, the solution’s portability is improved. Not only this, saving the abstract stack is less

expensive than performing a brute-force blind copy of the native stack. Finally, the performance of the examples presented by the authors resembled that of a pure event scheduling simulation.

Listing 2.3. Example of applying stack reconstruction (adapted from Perumalla and Fujimoto (1998))

```
method one() {
    statement1;
    if (...) {
        statement2;
        wait(cond1);
        statement3;
    }
    for (...) {
        statement4;
        two();
        statement5;
    }
}

method two() {
    ...
    wait(cond2);
    ...
}

main {
    ...
    one();
    ...
}
```

Listing 2.4. Method “one” from the previous listing with stack reconstruction (adapted from Perumalla and Fujimoto (1998))

```
method one() {
    switch(jumpIndex) {
        case 0: goto START;
        case 1: goto LBL1;
        case 2: goto LBL2;
    }
START:
    statement1;
    if (...) {
        statement2;
        wakeup = cond1;
        jumpIndex = 1;
        return SUSPENDED;
LBL1:
        statement3;
    }
    for (...) {
        statement4;
LBL2:
        flag = two();
        if (flag == SUSPENDED) {
            jumpIndex = 2;
            return SUSPENDED;
        }
        statement5;
    }
    jumpIndex = 0;
    return DONE;
}
```

2.7. Converting Process Interaction into Event Scheduling

The advantages of using process interaction, like being intuitive and providing a direct link to object orientation, were introduced earlier. Event scheduling is known for being the most efficient world view, but not at all easy to use, scalable or intuitive. However, Figure 1.1 already highlighted the relationship between both approaches, that is, a process can be easily seen as a succession of events. Consequently, the code executed between two consecutive suspensions of a process can be mapped to an event. Thus, each event is only responsible for scheduling the next piece of the process, which would be, in turn, a new event.

2.7.1. Manual Implementation

A process interaction model can be manually rewritten into an event scheduling one. This task can be carried out by detecting the conditional and unconditional delays of the process and placing the code up to the next delay in a new event. This procedure has several drawbacks:

- A comprehensive review of the previously created model is required;
- it is error prone;
- the semantics of the process recursion are difficult to capture by using events;
- local variables require special attention.

In the end, the main question is why an event-oriented simulator was not built in the first place.

2.7.2. Automated Translation

Having a more automated mechanism to carry out the transformation from process orientation to event orientation would be highly desirable. Possible solutions include using an interpreter or pre-compiling the model by applying some

kind of transformations to the source code. An automated translation could certainly suffer from some of the same drawbacks as a manual one; that is, local variables remain a problematic issue and recursion is difficult to handle.

When implementing a process-oriented simulation in a sequential computer, this is often the preferred approach since event scheduling is more efficient than process interaction, and the transformation itself can be, in some cases, almost trivial.

2.8. Java for Process-Oriented Simulation

Up to this point, the main techniques used to implement a process-oriented simulation tool have been reviewed, with Java™ being the target language for many of the examples presented. Java is an object-oriented programming language, originally developed by Sun Microsystems, with a syntax derived from C and C++. Generally, a Java program is not directly compiled into machine code. Instead, an intermediate representation termed *bytecode* is used. *Bytecode* is stored in *.class* files and run on a Java Virtual Machine (JVM).

In essence, Java defines four types of components, organised into *packages*: classes, interfaces, enumerated types and exceptions.

- *Classes* are the blueprints from which object instances are created. A class comprises a collection of fields and methods that define the properties and capabilities of an object.
- An *interface* is a list of methods to be implemented. Hence, it simply works as a statement of intent. Though Java does not permit multiple inheritances, a class can *implement* more than one interface.
- *Enumerated types* are a set of named values. Since Java treats enumerated types as a special compiler-generated class, they can declare instance methods and fields.

- *Exceptions* are special classes that handle the occurrence of extraordinary events or errors during the execution of a program.

The remaining of this section will survey the main reasons supporting Java as a suitable language for implementing the process interaction approach.

2.8.1. Object Orientation

Object Orientation (OO) has traditionally been put forth as a highly suitable paradigm for supporting DES (Pidd, 1995). Indeed, the first object-oriented language, SIMULA (Dahl and Nygaard, 1966), was developed to create and run simulation models. Furthermore, Fujimoto (1993) insists on the similarities between PDES and OO.

2.8.2. Portability

“Write once, run everywhere.” Sun Microsystems’ slogan is perhaps Java’s winning hand. Almost any system has a corresponding JVM: Linux, Windows, Solaris, Mac OS, HP-UX, cell phones, routers, washing machines. . . Portability means that a developer can focus on high-level characterisation and ignore hardware constraints such as word size, register file size and operating system.

2.8.3. Lack of Pointers: the Garbage Collector (GC)

Java avoids explicit memory management by the user. Thus, memory leaks and bad address arithmetic completely disappear, and Java stands as a safer language than, for example, C or C++. Not only this, the lack of pointers improves the security of the language. The automatic mechanism used for handling memory, the GC, allows the compiler and JVM to be further optimised.

2.8.4. Multithreading

Java threads are built directly into the language. Actually, Java core support for threads can be considered as simple and elegant, but also quite rudimentary. However, Java 1.5 included an extended package, `java.util.concurrent`, with thread pools, a framework for asynchronous task execution, *collection* type classes optimised for concurrent access, synchronisation utilities such as semaphores, atomic variables, blocks and condition variables (Lea, 2005). Such an array of features enables the user to more easily exploit multithread programming.

2.8.5. Network Aware

Java was created with network communications in mind. Several features can be highlighted that make Java network aware:

- Java supports network communication via *sockets* and provides support for higher level protocols, such as HTTP and FTP.
- *Remote Method Invocation* (RMI) is an extra abstraction layer over sockets. RMI allows a method of a class located on a remote computer to be invoked almost transparently from another computer. Since RMI is sometimes regarded as inefficient and unnecessarily complex, Philippsen et al. (2000) presented an alternative design called *KARMI*, which also includes an alternate serialisation mechanism.
- Java objects can be *serialised*, that is, disassembled, sent through a communication stream, and finally assembled again.
- Java supports the execution of small applications that can be invoked from a web browser. Despite the fact that *Applets* are still a widespread option for developing client front-ends, the *Java Web Start* (or *javaWS*) framework is gaining ground.

Chapter 3

SIGHOS: a Process-Oriented Simulator

Managers of complex organisations face the constant challenge of making strategic decisions that require a holistic view of their companies. Nowadays, the use of sophisticated information systems allows to better exploit the organisation's data to convert it into useful information. However, most decisions demand a global evaluation of the Business Processes (BPs) and a comprehensive review of the resources and entities that interact within the organisation. Modelling and simulation provide a powerful tool to analyse, understand and, eventually, change an organisation's processes so to improve its performance while reducing costs.

Though Business Process Simulation (BPS) (Wynn et al., 2008) represents an invaluable instrument to support the modelling and simulation of BPs, suitable tools (and experts) are required to fully benefit from this technique. The rest of this chapter goes through the basics of business process modelling and simulation. Furthermore, *SIGHOS*, a generic process-oriented simulator, developed in Java for the purpose of simulating organisations, is described.

3.1. Basic Definitions of Business Processes

The first step to take advantage of a business-centred view of an organisation is to accurately model the business processes. A comprehensive *process definition* includes all the relevant *subprocesses* and, ultimately, *activities*; their relationships; and the criteria to indicate the start and the end of the process.

However, a process definition is nothing but a static view of the organisation. In order to profit from Information Technologies (IT), such a definition should be integrated into an automated tool that is able to “execute” and monitor the BPs. Therefore, the Workflow Management Coalition (WfMC) defines the concept of *workflow* as:

« The partial or total automation of a business process during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules. »

Having defined what a workflow is, a *workflow management system* would be such an automated tool, intended to interpret process definitions, interact with workflow participants, and even invoke IT tools and applications. A workflow management system creates *instances* of the processes and activities described in the process definition. An instance is defined by the WfMC as:

« The representation of a single enactment of a process, or activity within a process, including its associated data. Each instance represents a separate thread of execution of the process or activity, which may be controlled independently and will have its own internal state and externally visible identity, which may be used as a handle, for example, to record or retrieve audit data relating to the individual enactment. »

Within the process definition, an *activity* is considered an atomic piece of work or logical step of the process, and generally requires human and/or machine resources. The WfMC distinguishes between automated activities, which are

executed by the workflow management system, and manual activities, which are out of the workflow management system's scope and are included for modelling purposes only.

Figure 3.1 shows the relationship among the different workflow concepts.

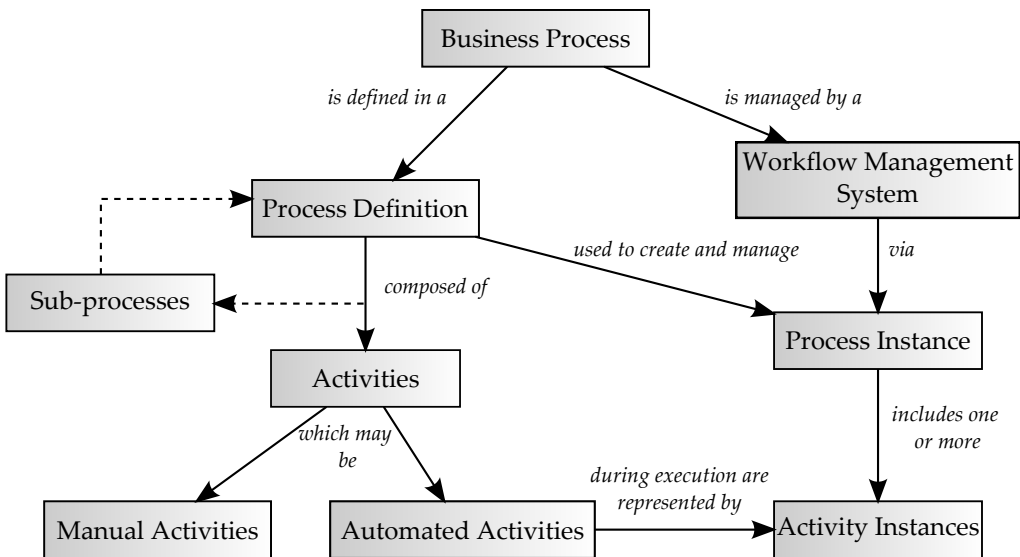


Figure 3.1. Basic workflow terminology. Adapted from (WfMC, 1999)

3.2. Workflow Patterns

Van Der Aalst et al. (2003) explain how there is little consensus in the workflow specification due to the lack of universal concepts for modelling business processes. This, according to them, accounts for most of the differences between the workflow languages. There are different perspectives to the workflow specifications:

- The *control-flow* perspective describes the activities and their execution ordering through different constructors.

- The *data* perspective layers business and processing data on the control perspective.
- The *resource* perspective provides an organisational structure anchor to the workflow in the form of human and device roles responsible for executing activities.
- The *operational* perspective describes the elementary actions executed by activities, where the actions map onto underlying applications.

From the point of view of Business Process Reengineering (BPR), the control-flow perspective offers the best opportunities for use in a simulation project. The data perspective basically rests on the control-flow one, whereas the resource and operational perspectives are ancillary.

As stated by Kiepuszewski et al. (2003), the basic control-flow constructs are *sequence*, *iteration*, *split* (AND and OR) and *join* (AND and OR). The concepts of sequence and iteration are trivial; however, split and join involve the appearance of multiple concurrent execution threads for a process instance:

- *AND-Split* is a point within the workflow where a single control thread splits into two or more threads that are executed in parallel.
- *AND-Join* is a point in the workflow where two or more parallel executing tasks converge into a single common control thread.
- *OR-Split* is a point within the workflow where a single control thread decides on which branch to take when faced with multiple alternative workflow branches.
- *OR-Join* is a point within the workflow where two or more alternative task workflow branches re-converge to a single common task as the next step within the workflow.

The basic workflow constructs are useful but their expressive power is limited. Thus, Russell et al. (2006) propose 43 control-flow patterns (numbered WCP1

– 43) that delineate the fundamental requirements for modelling the different scenarios defined within the control-flow perspective. These patterns are based on a study of countless practical cases involving real companies, and establish both a language- and technology- independent framework to describe, analyse and compare different workflow languages and tools.

Generally, workflow patterns are organised into several categories:

- *Basic control flow patterns* capture elementary aspects of process control, such as sequence of activities, parallel branches and conditional nodes. Basically, these patterns comprise the basic constructs mentioned above.
- *Advanced branching and synchronisation patterns* complete the basic ones with more complex and flexible branch and join structures.
- *Multiple instance patterns* involve activities that are able to initiate multiple instances of themselves when invoked. Some of the patterns already introduced in former categories may concurrently run multiple instances of the same activity, but these specific patterns are different since they usually require further synchronisation.
- *State-based patterns* reflect situations where the current system state is used to make a decision regarding the way the workflow is executed.
- *Cancellation and force completion patterns* refer to situations where one or several activity instances (or even the entire workflow) are withdrawn. Handling of exceptions relies on these patterns too.
- *Iteration patterns* capture repetitive behaviours.
- *Termination patterns* represent the different circumstances under which a workflow may be considered to be completed.
- *Trigger patterns* stand for tasks that are started when a certain external signal is received.

Appendix E describes each of the 43 patterns individually, whereas (Workflow Patterns Initiative, 2010) offers the most updated information and interactive contents.

3.3. SIGHOS: A Process-Oriented Simulation Tool

The Simulation Group from the Department of Systems Engineering and Automation (SIMULL) has been working on business process simulation for many years. Originally, their efforts were focused on the simulation of hospitals. Thus, a specific tool, SIGHOS (Muñoz and Castilla, 2010), was designed. SIGHOS stands for the Spanish *Simulación para la Gestión HOSPitalaria*, that is, Hospital Management Simulation. Soon, it became clear that the main ideas applied to simulate a hospital could be generalised and extended to deal with other organisations, such as call centres. Moreover, by adopting workflow patterns to model and simulate business processes, the scope of the tool significantly increased.

The rest of this section presents the main features and modelling capabilities of SIGHOS. Most of the concepts are adaptations of those defined in Chapter 1, and will be introduced by abstracting the final implementation in Java. Nevertheless, the translation from those definitions into a practical implementation is trivial.

3.3.1. Simulation

The first and most important modelling component of SIGHOS is `Simulation`.¹

Definition 3.1. Simulation

A `Simulation` $S(t_{end}, R, RT, A, WF, G)$ is a container for the rest of the modelling components, as well as the framework for performing the simulation itself. A `Simulation` has an associated finalisation timestamp t_{end} , and comprises a set of

¹While a more suitable name might be *Model*, for “historical” reasons, we have preserved the `Simulation` class as the model *container*.

Resources R , Resource Types RT , Activities A , Workflows WF and Element Generators G .

3.3.2. Resources

SIGHOS uses Resources to represent any human or material assets needed to carry out a task. Simulation tools do not, in general, treat resources as individual entities of the model. For example, any level 1 operator in a call centre can answer a call. What tools like Arena (Kelton et al., 2002) do is to define *roles* or Resource Types and then, for each of these roles, describe a function of the number of time-dependent Resources available. SIGHOS takes a completely different approach by calculating the availability function for each role dynamically based on each individual Resource's availability function. This availability is defined by a set of triplets $\langle c_i, ta_i, rt_i \rangle$ referred to as Timetable Entries, as shown in Figure 3.2.

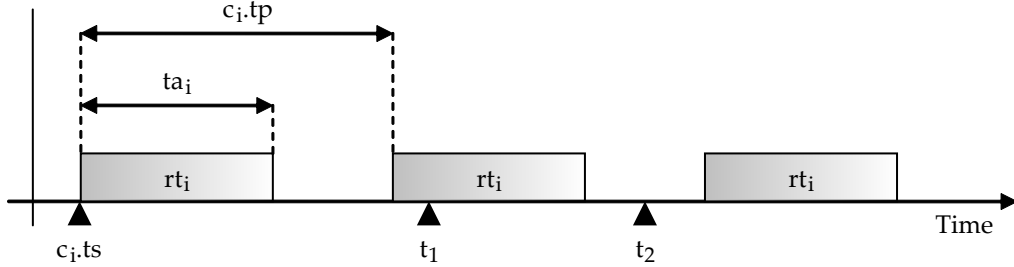


Figure 3.2. Resource availability for a Timetable Entry $\langle c_i, ta_i, rt_i \rangle$. The Resource is available as rt_i at t_1 but not at t_2 .

Definition 3.2. Timetable Entry

A Timetable Entry $tte\langle c_i, ta_i, rt_i \rangle \in r$ indicates that Resource r will be available during time period $ta_i \geq 0$ for role rt_i , following the time pattern indicated by c_i .

c_i is normally defined as a cyclic time pattern:

Definition 3.3. Cycle

A `Cycle` $c(t_{start}, t_{end}, t_{period})$ is a time pattern defining a time interval that starts at timestamp $t_{start} \geq 0$, ends at timestamp $t_{end} > t_{start}$, and defines the occurrence of an event with period t_{period} ($0 \leq t_{period} \leq t_{end} - t_{start}$). A `Cycle` $c(t_{start}, k, t_{period})$ can be also defined as a time interval that starts at timestamp $t_{start} \geq 0$ and which, for k iterations, defines the occurrence of an event with period t_{period} ($0 \leq t_{period} \leq t_{end} - t_{start}$).

Definition 3.4. Timestamp included in a Timetable

A `Timetable Entry` sets the start and end timestamps of each availability period of a `Resource`. A timestamp ts is said to be *included* in a `Timetable` $tte_n \langle c_n, ta_n, rt_n \rangle$, $c_n = (t_{start}, t_{end}, t_{period})$ iff $(t_{start} + k * t_{period} \leq ts < t_{start} + k * t_{period} + ta_n) \wedge (ts \leq t_{end})$. The inclusion operation will be denoted as $ts \subset tte_n$.

Definition 3.5. Resource availability

A `Resource` r is available for a `Resource Type` rt_a at simulation time t iff a `Timetable Entry` $tte_n \langle c_n, ta_n, rt_n \rangle$ exists such that $t \subset tte_n$. $|rt_n|(t)$ will denote the number of `Resources` available for role rt_n at timestamp t .

The availability of a `Resource` describes not just the timetable, but the function performed by such `Resource` in that time period. In other words, the same `Resource` can have different roles at different times, and may even play more than one role at a time. For example, a surgeon in a hospital can operate in the mornings (role: surgeon), and see patients as a doctor in the afternoons (role: doctor).

`Timetable Entries` allow a user to express complex temporal behaviour for the availability of a `Resource`. However, certain circumstances require additional mechanisms to model *unavailability*. For example, failures of a machine, illness of a human resource, an irregular work schedule, etc. are concepts that are difficult to introduce in a regular availability time pattern, and which can be better identified and expressed by using a specific unavailability or *cancellation*

time pattern. A Cancellation Entry $cte\langle c_i, ta_i, rt_i \rangle$ is normally defined together with its counterpart availability pattern and completes the previous definition of *Resource availability*.

Definition 3.6. Resource availability

A Resource r is available for a Resource Type rt_a at simulation time t iff $(\exists tte_n\langle c_n, ta_n, rt_n \rangle / t \subset tte_n) \wedge (\forall cte_m\langle c_m, ta_m, rt_m \rangle / rt_n = rt_m \Rightarrow t \not\subset cte_m)$.

It is important to note that a Cancellation Entry will always override any Timetable Entry making reference to the same Resource Type.

Obviously, the way SIGHOS handles the Resources adds additional complexity to the control of the simulation but, in exchange, it allows for the behaviour of Resources to be modelled much more accurately. When modelling the Resources of a complex organisation, such as a hospital, individual modelling provides two advantages:

1. It makes for a more direct translation from reality to simulation;
2. and it allows the schedule timetables to be adjusted, much as the manager of the organisation would do.

3.3.3. Activities

SIGHOS uses the definition of activity given in Section 3.1. However, the interaction between Activities and Resources requires some additional definitions.

Since any Activity usually requires more than one Resource to be carried out, the concept of Workgroup needs to be introduced.

Definition 3.7. Workgroup

A Workgroup $WG(\{\langle rt_1, k_1 \rangle, \langle rt_2, k_2 \rangle, \dots, \langle rt_n, k_n \rangle\})$ is a set of pairs $\langle rt_i, k_i \rangle$ where rt_i is a role and $k_i > 0$ is the number of Resources of that Type.

Definition 3.8. Workgroup availability

A Workgroup WG is available at time t iff $|rt_i|(t) \geq k_i \forall \langle rt_i, k_i \rangle \in WG$.

Thus, a more usable definition of Activity can be given:

Definition 3.9. Activity

An Activity $a(\{\langle WG_1, t_1, pr_1 \rangle, \dots, \langle WG_m, t_m, pr_m \rangle\})$ is a task that can be solved in time $t_i \geq 0$ if $WG_i, 1 \leq i \leq m$ is available.

The use of different Workgroups in an Activity allows, for example, for the “operation” Activity to be carried out with a Workgroup consisting of a surgeon, two assistant surgeons and two nurses and to last 1 h. The same Activity could be performed by one surgeon and two nurses and last 2 h. The different triplets $\langle WG_i, t_i, pr_i \rangle$ may be ordered in non-decreasing order and assigned a priority pr_i , such that the Activity is carried out by using WG_i available in time t iff $\nexists k < i / WG_k$ available at time t .

Definition 3.10. Feasible Activity

Activity a is said to be *feasible* at time t iff $\exists \langle WG_i, t_i, pr_i \rangle \in a / WG_i$ is available at time t .

3.3.4. Elements

Elements represent the main actors of the simulation, that is, the entities that *actively* interact with the system by carrying out Activities. Patients in a hospital, calls in a call centre, documents in a document management system... they can all be considered Elements of a simulation.

Linking with the definitions of the WfMC, an Element $e(wf)$ can be considered a process instance of the Workflow defined in wf . Thus defined, the life cycle of an Element $e(wf)$ consists of carrying out all the Activities referenced in wf , as seen in Figure 3.3. When there are not enough Resources to carry out an Activity, the requesting Element is enqueued until new Resources are available.

Each Activity implicitly includes an Element queue.

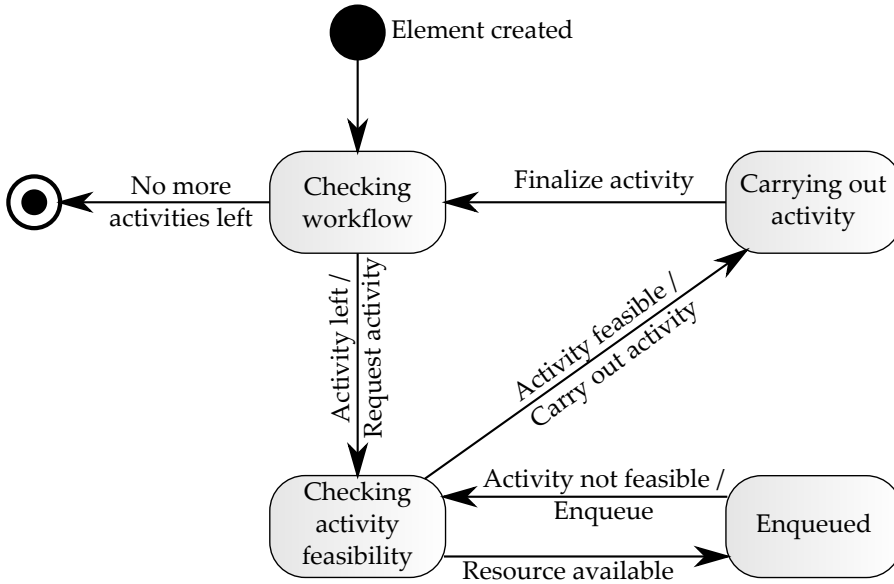


Figure 3.3. Life cycle of an Element

Definition 3.11. Carrying out an Activity

Consider an Activity $a(\{\langle WG_1, t_1, pr_1 \rangle, \dots, \langle WG_m, t_m, pr_m \rangle\})$ and an Element $e(wf) / a \in wf$. $e(wf)$ is said to carry out the Activity a with a Workgroup $WG_i (i \in 1 \dots m)$ iff a is feasible with WG_i and $e(wf)$ acquires as many Resources as are indicated by WG_i during a simulation time interval t_i .

Elements are classified into several Element Types (et_i). The reason is twofold: on the one hand, Element Types can be useful for statistical purposes when collecting the simulation results, and simply obey the logical structure of the system being modelled; on the other hand, different priorities can be associated with each Element Type. Priorities serve to create urgent Elements, for example, patients with a more serious illness, or incidents that should be treated promptly.

The workflow patterns introduced in Section 3.2 constitute the basis for the Workflow constructs in SIGHOS. This topic will be thoroughly explained in Section 3.5.

Elements appear in the system following a temporal pattern or in response to a simulation event. Thus, a modeller can define both time-driven and condition-driven Element Generators. Time-driven Generators can be used to model the arrival of patients at a hospital, which adheres to a time pattern. Condition-driven Generators can be used, for example, to model the arrival of incidents at a customer service centre. Although most of the user incidents will adhere to a time pattern, some may give rise to new incidents. For example, the receipt of many incidents of users being unable to access Internet could lead to a “check organisation’s network” incident.

When and *how many* are questions that are treated separately when generating Elements. Element Generators answer the first question (*when*), whereas the second (*how many*) relies on a different structure, usually called a *creator*. A creator consists of a quantity N , which can be fixed, random and even clock-related; and a set of triplets $(\langle et_1, wf_1, p_1 \rangle, \langle et_2, wf_2, p_2 \rangle, \dots, \langle et_k, wf_k, p_k \rangle)$, $\sum_{i=1}^k p_i = 1$. Every time the Generator is invoked, N new Elements are created based on the triplets. $p_i, i \in [1, k]$ denotes the probability of creating an Element belonging to Type et_i and carrying out a Workflow wf_i .

3.4. Inside SIGHOS: an Event Heart

One of the main objectives of SIGHOS is to make modelling easier by isolating the modeller from the implementation details of the simulator. Consequently, most of the design decisions of this simulator take into consideration this principle. However, it is undeniable that the implementation details of a simulation tool may dramatically affect the results obtained, as shown in (Schriber and Brunner, 1994) and later in (Schriber and Brunner, 2006). Actually, the importance of this topic is assessed by the selection of the first paper as one of the ten Landmark

Papers of the 40th anniversary of the Winter Simulation Conference in 2007.

In this and subsequent sections we will go through the implementation details of SIGHOS, which may impact the performance and results of executing simulation experiments.

3.4.1. First Approach: Threads

Being a process-oriented simulator, and having selected Java as the programming language, the first approach for implementing SIGHOS was to use a direct translation process – thread. Therefore, by using the most natural expression of a process-oriented simulation, the objective of isolating the modeller from the implementation details would be easily fulfilled.

The problems with this approach, as previously explained in Section 2.2, also apply to this case. Specifically, the kind of systems that SIGHOS is targeted at (that is, organisations such as hospitals) may involve thousands of entities simultaneously interacting (e.g. patients in a hospital). The limit to the number of threads that many operating systems impose prevents this solution from being considered a practical choice for simulating a model of an organisation.

Furthermore, as stated before, several workflow patterns involve the use of more than one thread of control per entity. Consequently, the number of threads required to implement a direct translation would quickly exhaust the system resources.

3.4.2. Using Event Scheduling

After discharging direct translation, a different approach is required. Such an approach must preserve the independence of the modeller from the simulation engine, while at the same time facilitating the adaption to a parallel execution. Therefore, SIGHOS is based on an automated translation into an event scheduling approach (see Section 2.7).

As seen in Subsection 1.2.1, the event scheduling approach defines an event ($e@ts$) as a set of actions to be performed at simulation time ts . SIGHOS establishes

a common framework for all the components of the simulation that are prone to creating events, named `Basic Element`.

In order to implement the event scheduling approach, SIGHOS requires a Virtual Clock (VC) and a FEL. The VC maintains the current simulation time (hereafter *CST*), and sets the timestamps for future events. The FEL contains a time-ordered list of the events with $ts > CST$. For simplicity, we will assume that there is a single `Simulation` object that contains such structures and defines two basic operations with `Events`:

- *schedule($e@ts$)*. Schedules an `Event` e to be executed at timestamp ts . If $ts = CST$, e is directly sent to be executed; whereas if $ts > CST$, the destination is the FEL. Events may be scheduled during the initialisation of the simulation or as part of the code associated with a former `Event`. An `Event` $e_1@ts_1$ cannot schedule an `Event` $e_2@ts_2$ ($ts_2 < ts_1$), thus preserving the causality constraint.
- *execute($e@ts$)* Executes the code or actions associated with the `Event` e . It may involve scheduling new `Events` or accessing shared simulation resources. We will pay special attention to the access to shared resources in Chapter 4.

3.4.3. Event Types Defined by SIGHOS

From the simulation components defined in Section 3.3, only `Resources`, `Elements` and `Generators` create events (and consequently extend the `Basic Element` framework).

Regarding `Elements`, the translation into an event scheduling approach takes advantage of the fact that SIGHOS describes the processes using predefined `Workflow` structures, whose leaf nodes are `Activities`. An `Element` behaves as a *process instance* and traverses its associated `Workflow`, as already shown in Figure 3.3. Every time an `Element` finds an `Activity` in the `Workflow`, the following `Events` are created:

- RequestActivity (Listing 3.1) checks whether an Activity is feasible. If so, the Element seises the corresponding Resources, computes a duration d for the Activity and launches a FinalizeActivity event with timestamp $CST + d$. Otherwise, the Element is queued in the Activity.

Listing 3.1. RequestActivity event

```

RequestActivity(Element  $e$ , Activity  $a$ ) {
  if ( $a.isFeasible(e)$ ) {
     $e.carryOut(a)$ ;
    Simulation.schedule(new FinalizeActivity( $e$ ,  $a$ ,
      Simulation.CST +  $a.duration$ ));
  }
  else {
     $a.enqueue(e)$ ;
  }
}

```

- FinalizeActivity (Listing 3.2) is launched when an Activity is finished. The Element releases the Resources previously took and continues with the subsequent Activities in the associated Workflow. Consequently, not only other Activities are notified of the newly available Resources, but additional RequestActivity events may be scheduled.

Listing 3.2. FinalizeActivity event

```

FinalizeActivity(Element  $e$ , Activity  $a$ ) {
   $a.releaseResources()$ ; // All the Activities using the
    released Resources are notified
  forall  $a_i \in$  (list of activities currently requested by  $e$ )
    do {
    Simulation.schedule(new AvailableElement( $e$ ,  $a_i$ ,
      Simulation.CST));
  }
  // Make  $e$  advance its associated workflow
  forall  $a_j \in$  (subsequent activities to be requested
    according to the associated workflow) do {

```

```
Simulation.schedule(new RequestActivity(e, aj,  
Simulation.CST));  
}  
}
```

- AvailableElement (Listing 3.3). Most process definitions may include cases when an Element is waiting in the queue of several Activities at the same time. Whenever such an Element completes an Activity, it has to inform the waiting Activities of its availability.

Listing 3.3. AvailableElement event

```
AvailableElement(Element e, Activity a) {  
    if (a.isFeasible(e)) {  
        e.carryOut(a);  
        Simulation.schedule(new FinalizeActivity(e, a,  
Simulation.CST + a.duration));  
        a.dequeue(e);  
    }  
}
```

Despite the fact that Elements are the main source of events, other simulation entities contribute with additional event types. Resources, due to the particular approach of SIGHOS, has to dynamically translate its availability, as defined in the Timetable Entries, into a set of RoleOn and RoleOff events. Let $\langle c, ta, rt \rangle / c = (t_{start}, t_{end}, t_{period})$ be a Timetable Entry for the Resource r (an analogous definition could be given for a Cycle $c' = (t_{start}, k, t_{period})$); the following events are created:

- RoleOn. An event of this type is scheduled at simulation time $t = t_{start} + t_{period} * k < t_{end}, k \in \mathbb{Z}^+$, making the Resource r available to role rt .
- RoleOff. An event of this type is scheduled at simulation time $t = t_{start} + t_{period} * k + ta < t_{end}, k \in \mathbb{Z}^+$, stopping the Resource r from being available to role rt .

These events tell the Resource Types to increase or decrease their available Resource counter, respectively. Moreover, any Activity using said Resource Type in any of its Workgroups is also notified in case of an increment. Hence, the Activity checks whether the new available Resources make it feasible for any of the Elements waiting in its queue.

SIGHOS defines two additional events (CancelOn and CancelOff) for the Cancellation Entries, analogously to the above RoleOn/RoleOff.

Figure 3.4 shows the life cycle of a Resource, according to the different events that may happen.

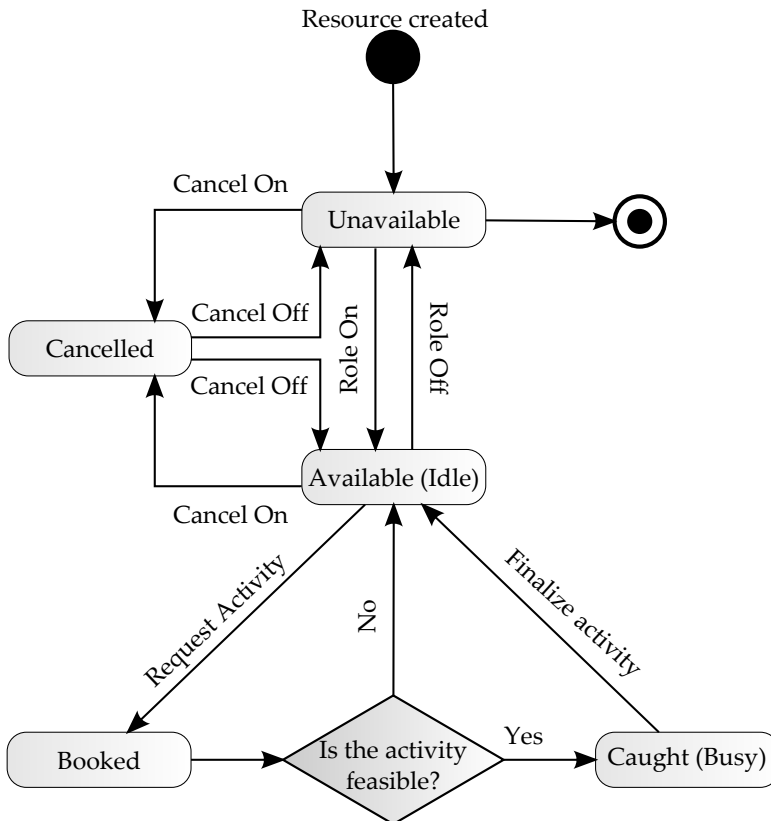


Figure 3.4. Life cycle of a Resource

Element Generators are the last source of events for the simulation. As previously explained in Subsection 3.3.4, Generators can react to changes in the system state, but are more frequently used to create Elements according to a time pattern. In this last case, the Generator invokes a Generate event following a time pattern defined by a Cycle c .

3.4.4. An Event Scheduling Algorithm for SIGHOS

Based on the event types and structures presented in the previous subsections, SIGHOS adapts the algorithm for event scheduling already introduced in Figure 1.2(a), as shown in Listing 3.4.

Listing 3.4. SIGHOS algorithm for event scheduling

```
// Phase 0: Initialisation: The initial events for the  
simulation entities are scheduled  
forall  $r_i \in \text{Simulation.R}$  do {  
    forall  $tte_j \in r_i$  do {  
         $\text{startTs} = tte_j.c.t_{\text{start}}$ ;  
         $\text{Simulation.schedule}(\text{RoleOn}_j@startTs)$ ;  
    }  
}  
forall  $g_i \in \text{Simulation.G}$  do {  
     $\text{startTs} = g_i.c.t_{\text{start}}$ ;  
     $\text{Simulation.schedule}(\text{Generate}_i@startTs)$ ;  
}  
while ( $\text{Simulation.CST} < \text{Simulation.t}_{\text{end}}$ ) {  
    // Phase 1: Update clock  
     $e_1@ts = \text{First event from FEL}$ ;  
     $\text{Simulation.CST} = ts$ ;  
    // Phase 2: Execution  
    forall  $e_i@ts / e_i \in \text{Simulation.FEL} \wedge ts = \text{Simulation.CST}$  do {  
         $\text{Simulation.execute}(e_i@ts)$ ;  
    }  
}
```

3.5. Inside SIGHOS: Managing Workflows

Events appear from a straightforward interpretation of the basic entities of the simulation. However, at a higher level of abstraction, the way workflow patterns are introduced in SIGHOS must be carefully considered since the behaviour of such patterns may be subtly modified depending on the specific implementation details.

3.5.1. A Review on Workflow Models

Kiepuszewski (2002) conducts a study on various commercial products and evaluates the different workflow modelling languages used, as well as their compatibility with the control-flow patterns. His study relies on the modelling technique used to generalise and classify the different models as follows:

- Standard Workflow Models
- Safe Workflow Model
- Structured Workflow Model
- Synchronising Workflow Models

Standard Workflow Models represent what would appear to be the most “natural” interpretation of the WfMC definitions. This kind of workflow model has no restrictions on the way the patterns are used. The result of this “freedom” is that arbitrary loops and multiple instances are allowed; by contrast, deadlocks may arise. Furthermore, the absence of strict rules describing the way to proceed when faced with possible deadlocks or infinite loops may result in ambiguity.

A Standard Workflow Model that does not allow multiple instances becomes a *Safe Workflow Model*. However, deadlocks are still possible.

Intuitively, a *Structured Workflow Model* is a model where each OR-Split has a corresponding OR-Join and each AND-Split has a corresponding AND-Join,

with no arbitrary cycles or multiple instances allowed, thereby avoiding deadlocks. At the same time, the expressive power of this kind of workflows is reduced.

Both Safe Workflow Models and Structured Workflow Models can be considered subsets of Standard Workflow Models. The use of the former in a simulation is discouraged since obviating the information from concurrent threads may affect the results of the running simulations. With respect to the latter, a structured framework makes the implementation easier, at the expense of reducing the expressive power of the models, which is highly undesirable.

Synchronising Workflow Models appear from a different interpretation of the WfMC definitions of basic control flow constructs. AND-Join and OR-Join are considered to typically follow an AND-Split and an exclusive OR-Split, respectively. Consequently, both AND-Join and OR-Join may be seen as constructs that synchronise a number of threads. In the case of the AND-Join, a number of *active* threads are synchronised; whereas in the case of the OR-Join, it may be considered that one active thread and several *inactive* ones are synchronised. A true/false token identifies whether a branch is active or not. Upon receipt of the token, each node in the model has to propagate a token indicating the validity of the process instance, true or false, to which the control flow is passed. This way, deadlocks are avoided. Unfortunately, according to Kiepuszewski et al. (2003), the formal definition of the Synchronising Workflow Models prevents a modeller from using multiple instances or representing arbitrary loops.

SIGHOS takes advantage of the token-based strategy introduced with Synchronising Workflow Models, but uses several strategies to relax the limitations regarding multiple instances and arbitrary loops that those workflow models impose.

3.5.2. SIGHOS and Workflow Patterns

SIGHOS defines a complex hierarchy of classes and interfaces that represent different behaviours from the workflow patterns, or a workflow pattern itself. All

of these classes include the suffix *Flow* in their names but, for the sake of simplicity, this suffix will be omitted hereafter. Figure 3.5 shows the basic classes. The first important distinction is made between *SingleSuccessor* and *MultiSuccessor* flows, that is, nodes allowing a single successor or several ones. Based on these two kinds of nodes, SIGHOS implements four basic constructs:

- Initializer node
- Finalizer node
- Task node
- Structured node

An *Initializer* node (see Figure 3.6) is a *MultiSuccessor* node that represents a source of new tokens, so that the generation of a token involves a new branch execution. *AND-Split*, *OR-Split* and *Conditional* nodes are based on this node. If a true token is received, the node propagates a true token through the active outgoing branches, and a false token through the inactive ones. In the event that a false token is received, the node propagates a false token through each outgoing branch.

Conditional nodes are a special case of *OR-Split*. Upon receipt of a true token, the node checks each associated condition, changing the resulting token value depending on the check result. Upon receipt of a false token, a false token is sent to each outgoing branch.

A *Finalizer* node (see Figure 3.7) is a *SingleSuccessor* node that represents a sink of tokens. A token removal represents the finalisation of an execution branch. *Finalizer* is the base for *AND-Join* or *OR-Join* nodes. The propagation of tokens depends on the criteria present at each node for yielding flow control.

- An *AND-Join* node generates a new token from the confluence of incoming tokens. This node defines an acceptance value as the number of true tokens that it must receive through each incoming branch. When enough tokens are received, a true token is sent to the node's successor and the

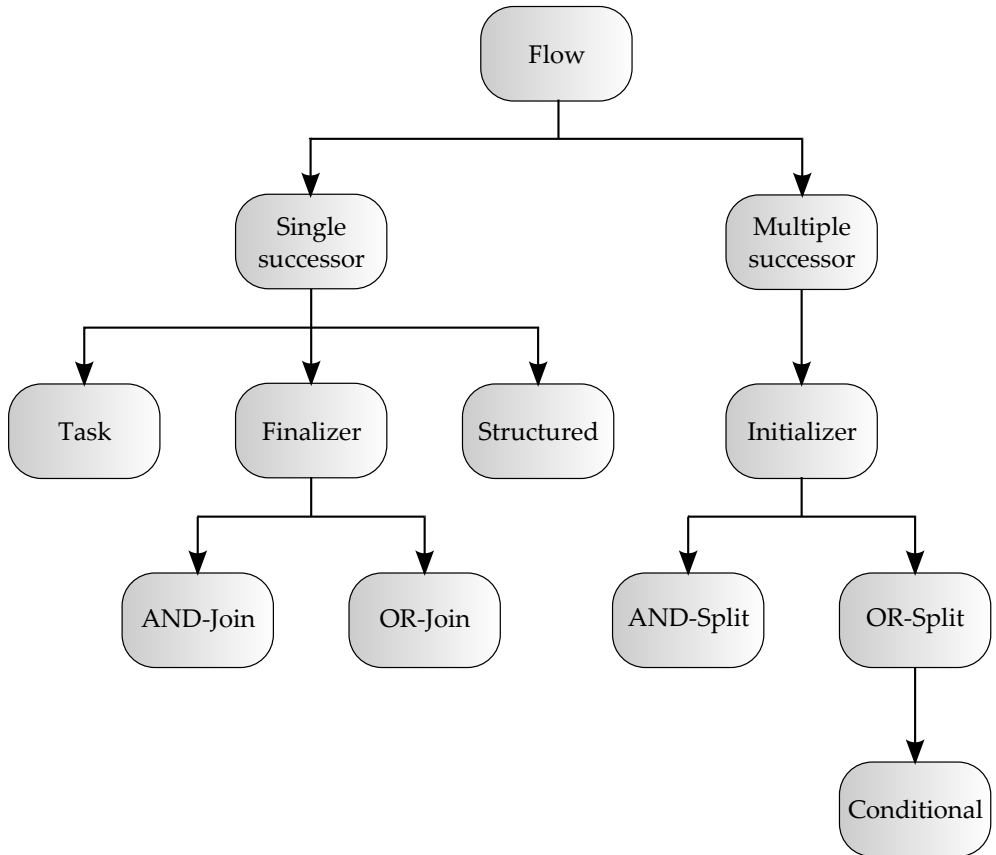


Figure 3.5. Basic workflow interfaces in SIGHOS

token count is reset. This type of node has two operating modes: *safe* and *unsafe*. Using one or the other operating mode affects the way the concurrent receipt of tokens through the same incoming branch is treated. In safe mode, only one token per branch and simulation timestamp is taken into account; the rest are simply discharged. In unsafe mode, all the tokens are considered, and taken as valid.

- An *OR-Join* node simply lets through the same token that arrived at the

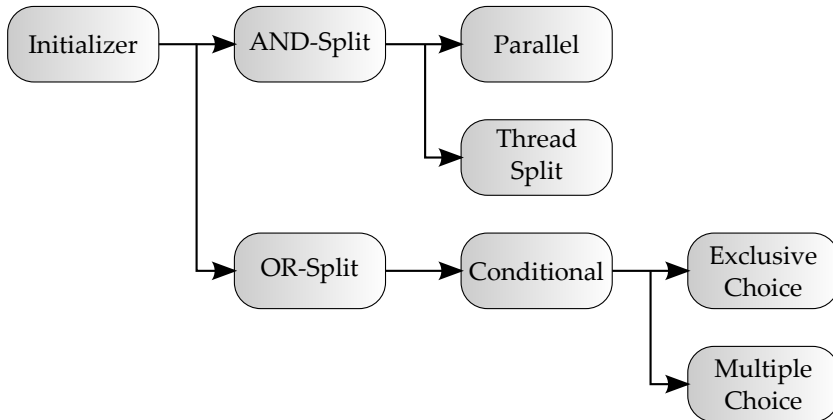


Figure 3.6. Initializer workflows

successor. The only control action that is taken involves the concurrent arrival of tokens. At that moment, the control state has to decide if all or only one thread is subsequently sent to the successors. If none of the incoming branches carries a true token, a false token is propagated.

A *Task* node is also a *SingleSuccessor* node, and represents the execution of an *Activity*. Using an explicit node to wrap an *Activity* is useful if the same *Activity* appears repeatedly in a *Workflow*. Being the same *Activity* means sharing the *Element* queue, but different user actions can be programmatically added depending on the specific step of the *Workflow* when said *Activity* is invoked. A task is executed upon receipt of a true token. Once the execution is complete, a true token is propagated again. However, if the execution is cancelled or the node receives a false token (meaning the execution is not to be carried out), a false token is propagated.

Structured nodes (see Figure 3.8) are *SingleSuccessor* nodes and consist of one node that defines the structure's starting point and another one that defines its end. All kinds of complex branches can be defined that link such starting and end points. The arrival of a false token implies the propagation of a false token

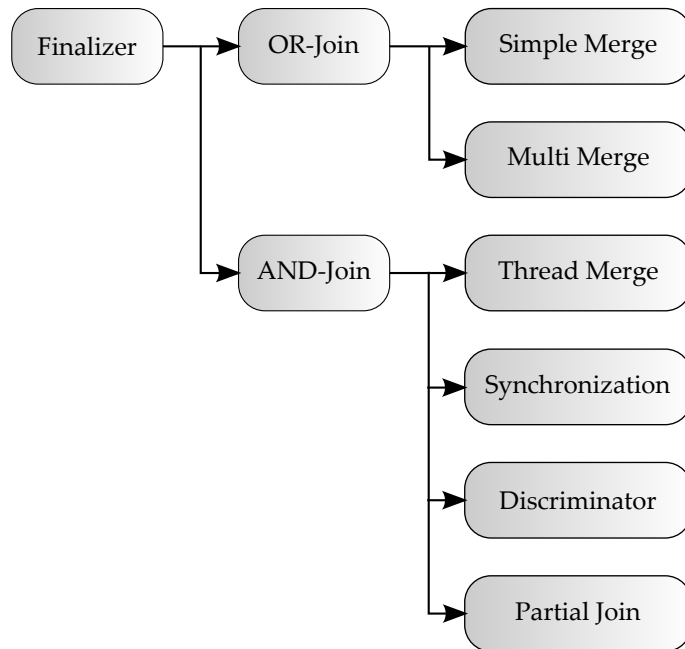


Figure 3.7. Finalizer workflows

without yielding control of the subnodes contained in the structure. If the opposite occurs, control is yielded to the sub-flow contained within the structural node.

Going back to Kiepuszewski et al. (2003), Synchronising Workflow Models were limited with regards to arbitrary loops and multiple instances. SIGHOS handles the patterns of arbitrary loops by implementing a system of environment variables, associated with the model or defined by the user, and an expression set that allows the conditions associated with these variables to be defined. There is also a set of user events that allows the values of these variables to be modified at different points in the model. These structures allow arbitrary cycles to be modelled, maintaining control over the possible appearance of infinite loops. When a true token enters one of these cycles, the set of conditions

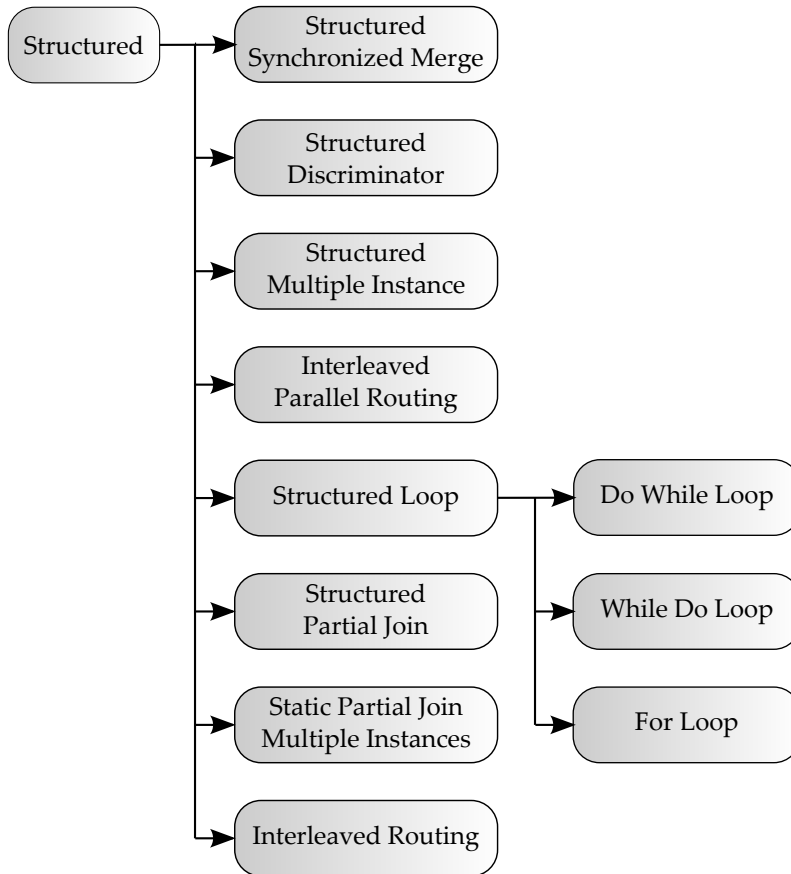


Figure 3.8. Structured workflows

associated with this cycle will control the exit from the cycle for that token. In contrast, when a false token enters one of these cycles, this token propagates, both to the exit branch for a cycle as well as to the branch that generates the loop. This leads to an infinite propagation of the false token, which results in an improper execution of the simulation. This undesirable situation can be solved if each false token keeps track of the nodes it has visited. Should a false token return to a node through which it has already passed, it is assumed to be immer-

sed in a loop and is deleted from the simulation since it is no longer producing relevant information.

Regarding multiple instances, SIGHOS defines *process threads*. A process thread represents a light process instance, in the same way that a thread represents a light process in an operating system. Process threads (implemented in a class `WorkThread`) are linked to the process instance and carry all the local information required for this execution branch, including local variables and even the true/false token. When a split node is encountered, a new process thread is created per outgoing branch, whether the branch is activated or not. A process thread, including a true token, would be created for the former, whereas a false one would be included in the latter.

Nevertheless, the real difficulties arise from those patterns that do *not* reflect multiple instances. These patterns, defined with Petri nets by Russell et al. (2006), have “safe places” where no concurrent process instances can exist. Several strategies have been developed to overcome this problem.

- There are patterns where safe places can be found before a node that represents the execution of an *Activity*. These places mean that said executions must be sequential, without concurrency for each process. SIGHOS allows the *Activities* to be categorised into two types: in person and in absentia. This concept comes from the modelling of hospital systems in which the patient, that is, the process instance, is faced with *Activities* that require his/her presence in order to be executed. This means that the patient is held by the *Activity* until its conclusion. Therefore, two instances of the same process that attempt to execute an in person *Activity* are forced to execute them sequentially.
- In the case of Join, Split or Structured nodes, the safe places indicate that the control of the branches leading to and from a node must be individualised for each process instance. To this end, specific control structures are implemented for each node.

To finalise this section, Table 3.1 shows whether SIGHOS supports each pattern or not.

Table 3.1. Support of control flow patterns

Pattern family	Pattern	
Basic Control Flow	WCP1: Sequence	✓
	WCP2: Parallel Split	✓
	WCP3: Synchronisation	✓
	WCP4: Exclusive Choice	✓
	WCP5: Simple Merge	✓
Advanced Branching and Synchronisation	WCP6: Multi-Choice	✓
	WCP7: Structured Synchronising Merge	✓
	WCP8: Multi-Merge	✓
	WCP9: Structured Discriminator	✓
	WCP28: Blocking Discriminator	✓
	WCP29: Cancelling Discriminator	✗
	WCP30: Structured Partial Join	✓
	WCP31: Blocking Partial Join	✓
	WCP32: Cancelling Partial Join	✗
	WCP33: Generalised AND-Join	✓
	WCP37: Local Synchronising Merge	✓
	WCP38: General Synchronising Merge	✗
	WCP41: Thread Merge	✓
WCP42: Thread Split	✓	

Table 3.1 – Continued

Pattern family	Pattern	
Multiple Instance	WCP12: Multiple Instances without Synchronisation	✓
	WCP13: Multiple Instances with a Priori Design-Time Knowledge	✓
	WCP14: Multiple Instances with a Priori Run-Time Knowledge	✗
	WCP15: Multiple Instances without a Priori Run-Time Knowledge	✗
	WCP34: Static Partial Join for Multiple Instances	✓
	WCP35: Cancelling Partial Join for Multiple Instances	✗
	WCP36: Dynamic Partial Join for Multiple Instances	✗
State-based	WCP16: Deferred Choice	✗
	WCP17: Interleaved Parallel Routing	✓
	WCP18: Milestone	✗
	WCP39: Critical Section	✗
	WCP40: Interleaved Routing	✓
Cancellation and Force Completion	WCP19: Cancel Task	✗
	WCP20: Cancel Case	✗
	WCP25: Cancel Region	✗
	WCP26: Cancel Multiple Instance Activity	✗
	WCP27: Complete Multiple Instance Activity	✗

Table 3.1 – Continued

Pattern family	Pattern	
Iteration	WCP10: Arbitrary Cycles	✓
	WCP21: Structured Loop	✓
	WCP22: Recursion	✗
Termination	WCP11: Implicit Termination	✓
	WCP43: Explicit Termination	✗
Trigger	WCP23: Transient Trigger	✗
	WCP24: Persistent Trigger	✗

From the set of patterns not yet supported, non-local semantics (Aalst et al., 2002) supposes the main source of problems. Non-local semantics make reference to those situations where a node requires information that is not local to the node (that is, which belongs to a different part of the model) in order to make a decision on the propagation of a process instance. The token strategy copes with some of the problems derived from such semantics, but others remain unsolved.

Cancellation patterns (WCP19, WCP20, WCP25, WCP26, WCP27, WCP29, WCP32 and WCP35) perfectly illustrate the problems derived from the use of non-local semantics. These patterns require explicitly breaking the execution logic of the simulator. A cancel pattern normally affects a process instance or a set of process instances. However, the process instance that activates the cancellation can be located in a completely different part of the model. Locating such instances implies complex memory structures and an efficient search algorithm, thus increasing the simulator memory and CPU requirements.

3.6. Inside SIGHOS: . . . Why It Matters?

As previously introduced in Section 3.4, the implementation details are important when trying to understand the specific behaviour of a simulation tool when

faced with different situations. This section will review the same questions posed in (Schriber and Brunner, 2006, 1994). These papers are based on the transaction-flow world view, which is considered a subtype of process interaction.

« In the transaction-flow world view, a system is visualised as consisting of discrete units of traffic that move (“flow”) from point to point in the system while competing with each other for the use of scarce Resources. The units of traffic are sometimes called *transactions*, giving rise to the phrase *transaction flow*. »

This definition is clearly in keeping with the objectives of SIGHOS, by exchanging *transaction* by *Element*. Therefore, the different components of this approach will not be described again, but the definitions introduced at the beginning of this chapter will be used.

3.6.1. Trying to Recapture a Resource Immediately

The first case, as stated in (Schriber and Brunner, 1994), makes reference to an *Element* that releases a *Resource* and then immediately attempts to recapture the *Resource*. The modeller might want - or not - a different *Element* (perhaps one with a higher priority) to capture the *Resource* next. There are at least three different alternatives to solve this situation:

1. The availability of the *Resource* is announced *before* the current *Element* is effectively considered a “contender.”
2. The availability of the *Resource* is announced only when the current *Element* can be considered a valid “contender.”
3. The current *Element* recaptures the *Resource* while ignoring the other “contenders.”

Resources are not directly “captured” in SIGHOS. Instead, *Elements* request *Activities*, which make reference to the *Resources* required to carry them out.

The case described here would be translated into an Element requesting two consecutive Activities, both of them making use of the same Resource. In such a case, the inner implementation obeys the first behaviour, that is, the rest of the “contenders” will be aware of the availability of the Resource before the current Element is effectively “freed.”

3.6.2. The First in Line is Still Delayed

The second case involves two Elements both requesting the same Resource Type. The first Element requests only one Resource, whereas the second one requests two. If no Resources are available, both Elements have to wait but, what happens when a single Resource becomes available? At least three courses of action are possible:

1. Neither Element uses the Resource.
2. The first Element “books” the Resource and waits for a second one.
3. The second Element captures the Resource.

Taking into account the use of Activities in SIGHOS, the problem formulation changes as follows: Consider an Element e_i , which is requesting an Activity a_i that requires one Resource of Type rt_a ; and a second Element e_j , requesting an Activity a_j that requires two Resources of Type rt_a . When a Resource r_k becomes available for rt_a at time ts , the behaviour of SIGHOS corresponds to the last course of action, that is, Resources required to carry out an Activity a are captured only when a is feasible; otherwise, the Resources remain idle.

3.6.3. Yielding Control

An Element that is currently being processed can *yield* control to other Element, and then take control again before the simulation clock advances. This can be useful if certain actions that affect other Elements need to be carried out before the current Element is completely processed.

From the point of view of a pure process-oriented approach, this case makes sense, since processes advance “as much as possible” (as seen in Subsection 1.2.3). Since SIGHOS transforms processes into a set of events, Elements can certainly yield control to other Elements when scheduling new events with delay 0. This feature is normally hidden from the final user but implicitly used, for example, when an Element is finalising an Activity and releasing Resources, as shown in the first case in this section.

The remaining cases in (Schriber and Brunner, 1994) are based on making Elements wait for the fulfilment of complex conditions. SIGHOS makes use of conditions as part of the Workflow definitions, but does not support trigger patterns. Therefore, only “wait for Resources” and “delay until time t is reached” are naturally implemented in the library.

Chapter 4

Parallel SIGHOS

As stated in the introduction, parallelism plays an important role in simulations due to the complexity of the models. Furthermore, a simulation, as used by a manager in an organisational environment, is intended to be integrated with other tools as part of the decision-making process. This use of a simulation generally involves many replications with different parameters. In this context, even with simple models, any reduction in the execution time of a simulation experiment would be valuable.

Unfortunately, despite being a valuable tool, parallelism is normally beyond the manager's scope. Modelling and simulation, in the end (and especially as expressed with SIGHOS), handle abstractions that can be easily mapped onto specific concepts that are familiar to a manager: processes, resources, activities... Contrarily, parallelism demands a strong background in computer science. Consequently, while *playing* with a simulation is a reasonable objective for a manager, *fine tuning* the parameters of a parallel simulation would clearly go beyond his / her skills.

Before analysing the possible alternatives for incorporating parallelism into SIGHOS, Table 4.1 enumerates the main goals of the intended solution, hereafter PSIGHOS (for Parallel SIGHOS).

Some additional explanations are required with respect to goal G4. When an algorithm that solves a specific problem is parallelised, the desired result is to obtain better execution time than the equivalent sequential algorithm. SIGHOS is

Table 4.1. Main goals of the intended solution

G1	The way the modeller uses SIGHOS must be preserved, that is, the modelling approach cannot change.
G2	Solutions from parallel and sequential SIGHOS for the same input model must be equivalent.
G3	The parallel implementation must be transparent to the user. Parallelism should be automatically exploited and not require any intervention from the user.
G4	The execution time of the parallel simulation must outperform that of the sequential one.

focused on a specific area (simulation of organisations) but not on a specific problem. This, together with the achievement of goals G1, G2 and G3, considerably increases the probability of finding systems that are mostly non-parallelisable. As a consequence, it would be possible to obtain an execution time for the parallel algorithm that is *worse* than the sequential one under certain circumstances. The key to this problem is to try to minimise the appearance of those cases by looking for a sufficiently flexible parallel solution; and to ensure that whenever the loss of performance is unavoidable, it is constrained to reasonable levels.

The rest of this chapter details the approach followed to exploit parallelism in SIGHOS so to take advantage from multi-core computers while fulfilling the goals stated in Table 4.1.

4.1. From Sequential to Parallel

In Section 1.3, we divided the main approaches to PDES into three groups depending on the extent to which the parallelism is exploited: application, simulation and model.

Application-level parallelism (which would include automatic code parallelisation, replicated trials and dedicated execution) clearly fulfils the previously

stated requirements. However, this solution neglects the high-level knowledge of the simulator (and certainly of the model), thus limiting the potential gain of this approach.

Contrarily, model-level parallelism fully exploits the parallelism present in the model. Thus, these techniques (cloning, domain decomposition and hierarchical decomposition) should result in the best performance. Unfortunately, these techniques require using characteristics that are specific to the model. A generic solution based on any one of these approaches would quickly degrade its performance.

Simulation-level parallelism, being half way between the two previous approaches, offers the best scenario for obtaining a generic parallel simulator that does not require changing the modelling approach. It also keeps the modeller from having to handle problems related to parallelism. We have noted several times over the course of this thesis that the FEL is one of the main bottlenecks of a DES. Perhaps the main proof supporting this statement is that the most popular approach to PDES, that is, domain decomposition, is specifically designed to create a partition of the FEL. While the performance of a centralised event list with distributed events would certainly never reach that of a specific solution created by using model-level parallelism, the possibility of reusing the same simulation engine would offset the moderate loss in performance.

The algorithm introduced in Listing 3.4 can be easily adapted to a centralised list with distributed events approach by using a *master-slave* (MS) paradigm. Hence, a master execution thread, or Event Manager (EM), executes the main simulation loop by updating the VC and distributing the current events into a set of slave worker threads, or Event Executors (EEs), which effectively execute the events. This means that there is a difference between sending or *dispatching* an event to be executed, and effectively *executing* such event. The EM is in charge of the former, and the EEs of the latter. The deferred event execution involves adding a mechanism to inform the EM that an event has been definitely executed. Listing 4.1 shows the modifications required to the EM algorithm so that it can handle the new parallel features.

Listing 4.1. Event Manager main loop (adapted from Listing 3.4)

```
// Phase 0: Initialisation: The initial events for the
// simulation entities are scheduled
forall  $r_i \in \text{Simulation.R}$  do {
  forall  $tte_j \in r_i$  do {
    startTs =  $tte_j.c.t_{start}$ ;
    Simulation.schedule(RoleOn $_j$ @startTs);
  }
}
forall  $g_i \in \text{Simulation.G}$  do {
  startTs =  $g_i.c.t_{start}$ ;
  Simulation.schedule(Generate $_i$ @startTs);
}
while (Simulation.CST < Simulation. $t_{end}$ ) {
  // Phase 1: Update clock
   $e_1@ts$  = First event from FEL;
  Simulation.CST = ts;
  // Phase 2: Execution
  forall  $e_i@ts$  /  $e_i \in \text{Simulation.FEL}$ 
   $\wedge$  ts = Simulation.CST do {
    ee = Next Available Event Executor;
    ee.dispatch( $e_i@ts$ );
  }
  // Synchronisation of Phase 2
  Wait until all the events have been effectively executed
}
```

Within this scheme, only the EM would remove events from the FEL; whereas the EEs would access and modify the simulation state and add new events to the FEL. Figure 4.1 illustrates the interactions among the EM, the EEs and the basic structures of the simulation engine.

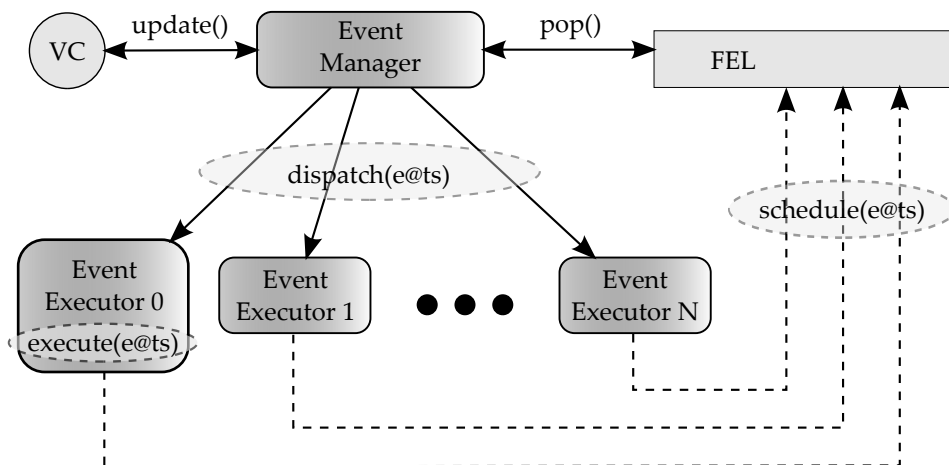


Figure 4.1. Basic schema of the Master-Slave approach

4.2. Limits to Parallelism: Resource Contention

In contrast to domain decomposition, which generally uses some kind of lookahead, our proposal is limited to executing the events corresponding to the current simulation time. Hence, this strategy imposes two obvious limits on the exploitable parallelism:

1. The average number of events with the same timestamp.
2. Concurrent events accessing the same simulation structures or components (Elements, Resources, Activities...).

With respect to the first limit, it is clear that the greater the number of events to be executed at the same timestamp, the more the parallelism that may be exploited. Certainly, models that are sufficiently large and complex should involve the execution of multiple events per timestamp.

Regarding the second item, resource contention (see Appendix F for an in-depth discussion on this topic) is a classical problem in the field of concurrency and parallelism, but one that has received surprisingly little treatment in the

PDES literature. The main reason can be attributed to the way a classical LP simulation works: each LP simulates a physical process and updates its state as it advances through time. Generally, LPs only communicate by exchanging messages. Within this scheme, access to resources is implicitly handled by the synchronisation algorithm.

Among the few references considering this problem, Reynolds (1982) proposes the Shared Resource Algorithm for Distributed Simulation (SRADS), later refined in (Nicol and Reynolds, 1985; Reynolds, 1983). This algorithm defines shared resources as communication channels among LPs and calls them Shared Facilities (SFs). As defined, an LP can access a SF if all the other LPs connected to this SF have clock values equal to or higher than the requesting LP. According to the authors:

« SRADS is best suited for applications where the relative frequency with which LPs must access SFs is low in comparison to the amount of time spent simulating non-communication related events. »

Consequently, for a resource-intensive simulation tool like SIGHOS, using SRADS would be counterproductive.

A more recent proposal, already cited in Section D.1, is (Cassel and Pidd, 2001). Although focused on domain decomposition, Cassel and Pidd substitutes the classical event orientation, which constitutes the basis for this kind of parallel technique, with a slightly more sophisticated three-phase approach. As previously explained in Subsection 1.2.4, the three-phase approach distinguishes between two kinds of events: *Bs* and *Cs*. Mapping *Cs* (or *conditional*) events onto a single LP is problematic, since they generally involve accessing more than one shared resource. Cassel and Pidd solve this issue by combining ideas from cellular simulation (Carvalho and Crookes, 1976), and push processing and replicated routes (Fatin, 1996).

- *Cellular simulation* starts by considering that not all *Cs* nor *Bs* require access to the whole set of shared resources. Therefore, *Cs* and *Bs* that access the same resources are grouped into “cells”.

- Fatin (1996) suggests that each LP should have a copy of the resources it shares with one or more LP(s); not only this, each message sent to an LP should be also sent to its “competitors”. The replicated routes serve to reduce communications but complicate the design of the LPs.

The result of linking both cellular simulation and Fatin’s work is the creation of a set of *CentralCs*. A *CentralC* combines a set of conditions (related to *C* events) and the resources required by these conditions into “cells”. LPs then communicate with these *CentralCs*, thus preventing the replication of resources and reducing the amount of messages required by Fatin’s proposal.

4.3. Resource Contention in SIGHOS

Subsection 3.4.3 described the event types that comprise the SIGHOS core. From those events, we can identify and enumerate the most relevant situations involving a concurrent access risk.

1. An Element may request / finalise several Activities
2. An Activity may be requested / finalised by several Elements
3. Several Resources are made / stop being available for the same Resource type
4. Several Elements requesting different Activities need the same Resource

Of all these problems, accessing Resources is the most challenging. Consider the following situation: Activity a_1 requires Resources of Type rt_1 and rt_2 , and Activity a_2 requires Resources of Type rt_2 and rt_3 . If a Resource with role rt_1 becomes available and is assigned to a_1 , and another Resource with role rt_2 becomes available and is assigned to a_2 , neither of the two Activities can be carried out. If, however, a different assignment is made, then at least one of the tasks can be executed. The example above can be further complicated by a circular wait resulting in a deadlock.

To avoid this problem, the Resource acquisition process is made in two stages: *book* and *capture*. In the first stage, the Element ensures that enough Resources are available to carry out the Activity; and, in the second, the Element actually acquires the Resources. This two-stage division is not arbitrary; rather, it allows for access to essential Resources to be restricted. Without this division, the entire set of simulation Resources would have to be blocked while an Element checked the feasibility of an Activity.

The way in which a Resource's timetable entries are defined determines the level of difficulty and, therefore, the complexity of the concurrent solution to be applied. Some additional definitions are needed to describe the various cases.

Definition 4.1. Timetable entry intersection (\cap)

Given two timetable entries tte_1, tte_2 , their intersection is defined as the time interval when both are simultaneously available, that is, $\forall t / t \subset tte_1 \wedge t \subset tte_2$.

Definition 4.2. Overlapping timetable entries

Two timetable entries tte_1 and tte_2 overlap iff $tte_1 \cap tte_2 \neq \emptyset$

Figure 4.2 is a schematic representation of two overlapping timetable entries for a Resource.

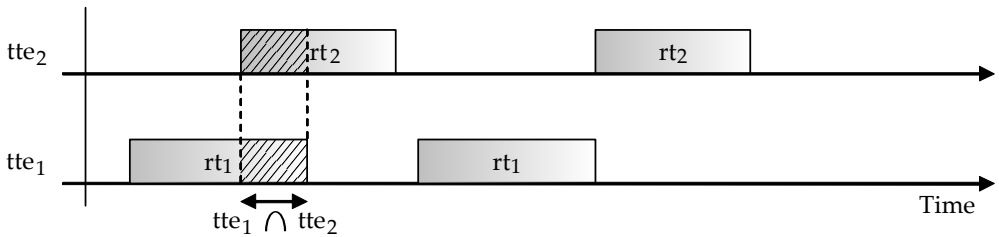


Figure 4.2. Overlapping timetable entries

4.3.1. No Overlapping Entries

Suppose only Resources $r\{\langle c_1, ta_1, rt_1 \rangle, \dots \langle c_m, ta_m, rt_m \rangle\}$ are used, such that none of their timetable entries overlap in time, that is:

$$\forall r\{\langle c_1, ta_1, rt_1 \rangle, \dots \langle c_m, ta_m, rt_m \rangle\} \Rightarrow \bigcap_{i=1..m} \langle c_i, ta_i, rt_i \rangle = \emptyset$$

In this case, a problem arises when several Activities require the same Resource Type in order to be carried out, which could result in a deadlock. This problem is solved through the use of structures called Activity Managers (AMs). An $AM_i(RT_i, A_i)$ is a partition in the set of Activities A and Resource Types RT in the Simulation, such that

$$\forall AM_i, AM_j / i \neq j \Rightarrow RT_i \cap RT_j = \emptyset \wedge A_i \cap A_j = \emptyset$$

In other words, the Activities belonging to a subset of this partition are independent from those in other subsets in terms of possible conflicts in Resource Type.

Though developed completely independently, this idea clearly parallels that of the *CentralCs* (Cassel and Pidd, 2001) defined in the previous section. The main difference lies in the parallel model used: message-passing in a distributed platform for *CentralCs* versus shared-memory for *AMs*.

The Activities and roles have to satisfy the following conditions:

1. If $WG_i \in a_i$ requires rt_k , and $WG_j \in a_j$ requires $rt_k \Rightarrow a_i$ and a_j must belong to the same AM.
2. $(\forall rt_i \in WG_j \in a_k) \wedge (a_k \in AM_l) \Rightarrow rt_i \in AM_l$.
3. Any modification to the Resource Types and Activities belonging to an AM must be mutually exclusive.

Constructing an $AM_i(RT_i, A_i)$ is equivalent to finding the connected components in a graph $G = (V, E)$ such that each vertex $v \in V$ is a Resource Type and

every edge $e \in E$ is an Activity that makes use of the Resource Types indicated at its vertices.

The following example, in which the Workgroups have been omitted for simplicity, clarifies the construction of this graph:

Suppose that $A = (a_1, a_2, a_3, a_4)$, $RT = (rt_1, rt_2, rt_3, rt_4, rt_5)$, and that the following associations between Activities and Resource Types hold: a_1 requires (rt_1, rt_2) , a_2 requires (rt_3, rt_4) , a_3 requires (rt_1, rt_4) , and a_4 only requires (rt_5) .

Firstly, a graph representing the associations is built (Figure 4.3). Then, this graph is split into connected components which can be mapped to obtain the following activity managers:

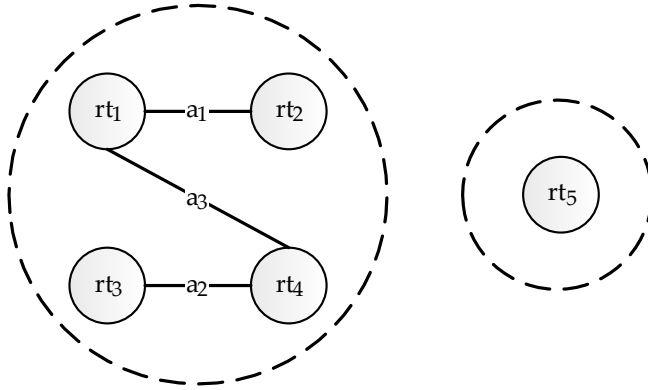


Figure 4.3. Construction of Activity Managers

- $AM_1(RT_1, A_1) \equiv RT_1 = \{rt_1, rt_2, rt_3, rt_4\}, A_1 = \{a_1, a_2, a_3\}$
- $AM_2(RT_2, A_2) \equiv RT_2 = \{rt_5\}, A_2 = \{a_4\}$

Each group of Activities and the associated Resource Types resulting from the partition is controlled by a different AM.

4.3.2. Overlapping Entries in Roles Belonging to the Same AM

Now consider a Resource r whose timetable entries overlap in time for Resource types belonging to the same AM, that is, $r\{\langle c_1, ta_1, rt_1 \rangle, \dots, \langle c_m, ta_m, rt_m \rangle\} / \forall rt_i \in AM_k, 1 \leq i \leq m$. Since all the roles belong to the same AM, the same control mechanism from the above case can be used. The possibility of having the same Resource appear more than once in the same Activity, however, presents a new problem, the solution to which has nothing to do with concurrence, but rather with the detection of this situation and its subsequent solution via an algorithm that allows for the available Resources to be optimally distributed among the different Resource Types requested by the Activity.

Given the nature of simulation problems, the definition of an Activity should not require a very high number of Resource Types. Further, we should not expect to have a very high number of Resources of each type. Consequently, a simple *Backtracking* algorithm can be adopted to solve the problem.

4.3.3. Overlapping Entries in Roles Belonging to Several AMs

In this case, the limitation in the above scenario disappears, that is, the roles for which a Resource can be available at the same time can belong to different AMs. Consider Figure 4.4 as an example. This figure shows two Elements, e_1 and e_2 , which request $a_1 \in AM_1$ and $a_2 \in AM_2$, respectively. Workgroups have been omitted for simplicity, the assumption being that each Activity requires just one Resource Type. Resource $r_1\{\langle c_1, ta_1, rt_1 \rangle, \langle c_2, ta_2, rt_2 \rangle\}$ has two timetable entries such that $\langle c_1, ta_1, rt_1 \rangle \cap \langle c_2, ta_2, rt_2 \rangle \neq \emptyset$, meaning it will be simultaneously available to both rt_1 and rt_2 . The access protection provided by the AMs is insufficient in this situation.

Since timetable entries are dynamic and can use any probability distribution in their definition, the simulation engine cannot know beforehand if a Resource will be available to an AM at a given timestamp. An extremely conservative solution would be to survey all of a Resource's timetable entries prior to commencing the simulation. Resource Types for which the Resource is made available

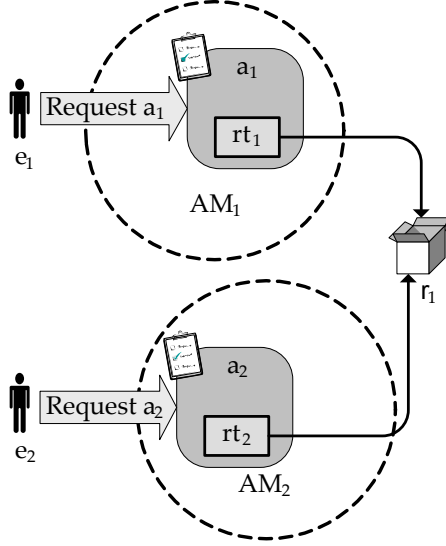


Figure 4.4. Resource simultaneously available for roles belonging to different activity managers

would be taken, and all the AMs containing those Resource Types combined. Hence, the AM's access control would act as a safety mechanism, the compromise being the sequencing of the simulation. The process we will use instead consists of dynamically isolating only truly conflicting Resources. To clarify this concept, let us establish the following definitions:

Definition 4.3. Potentially conflicting resources

A Resource $r\{\langle c_s, ta_s, rt_s \rangle, \dots \langle c_t, ta_t, rt_t \rangle\}$ has a *potential conflict* due to rt_s and $rt_t \Leftrightarrow (\langle c_s, ta_s, rt_s \rangle \cap \langle c_t, ta_t, rt_t \rangle) \neq \emptyset \wedge rt_s \in AM_j \wedge rt_t \in AM_k, j \neq k$.

Definition 4.4. Conflicting resources

A Resource $r\{\langle c_s, ta_s, rt_s \rangle, \dots \langle c_t, ta_t, rt_t \rangle\}$ has a *conflict* at simulation time t iif

- r has a *potential conflict* due to rt_s and rt_t
- e_1 requests Activity a_1 at simulation time t , and a_1 requires rt_s

- e_2 requests Activity a_2 at simulation time t , and a_2 requires rt_t

The last definition can be extended to regard e_1 and e_2 as having a conflict. Access to conflicting Resources must be arbitrated. Taking advantage of the two-stage Resource acquisition mechanism, the idea is to allow unrestricted access to Resources during the book phase. Then, and only if a conflict was detected during the first stage, would access to the Resources during the *capture* stage be arbitrated.

Book Phase

The conflicting Resources are identified during this phase. Each Resource has an associated booking list where the Elements that are simultaneously requesting an Activity that requires this Resource are inserted.

Initially, each Element is associated with a critical zone, which we will call Conflict Zone (CZ). A CZ contains a list of conflicting Elements, along with a synchronising mechanism that manages access to the zone itself.

Suppose an Element e tries to book a Resource r . The first thing to do is to check whether r 's booking list contains more Elements. If so, e 's CZ must be *merged* with that of the Elements in the list. The merge operation requires exclusive access to both CZs, meaning that, in order to avoid deadlocks, the merge is always done *from* the CZ of the higher index Element (or *source*) *to* that with the lower index (or *destination*). The merge operation adds all the Elements in the source CZ to that of the destination CZ. Once this is done, the Elements in the source zone are surveyed and targeted to the destination zone.

Capture Phase

During the capture phase, only one Element in the CZ should be able to acquire Resources at a time. Given what we have seen up to now, it would suffice to use a synchronisation mechanism associated with the CZ such that the Resource acquisition is mutually exclusive.

There is a serious drawback to this solution, since other Elements in the conflict zone may still be booking their Resources, meaning the composition of the conflict zone is susceptible to change with the addition of new Elements. Even that would not pose a problem as long as the new CZs being merged did not have any Elements already in their acquisition phase.

The problem with merging two conflict zones is what to do with the synchronisation mechanisms being used in each zone. Given that the goal is to have the synchronisation encompass the new merged zone without continuing to maintain the synchronisation independently for each subset of Elements, we propose the use of a structure comprising a stack of synchronisation mechanisms. Each CZ starts out with an empty synchronisation stack during the booking phase. When the first Element arrives at the capture phase, a synchronisation mechanism is added to the stack. Any new Elements arriving in this CZ during the capture phase will use this mechanism.

If the current CZ has to be merged with another one, the stack from both zones will be merged as well. If no Element in the second CZ has reached the capture phase, the merge does not alter the stack. Contrarily, if the stack in the second zone is not empty, the synchronisation mechanisms from this zone are added to those in the first.

Whenever an Element in the CZ reaches the capture phase, it receives a copy of the synchronisation stack and surveys the mechanisms contained within it until exclusive access to the Resources can be guaranteed. When the Element completes the capture phase, it uses its copy of the stack to release said mechanisms in order. The use of a copy instead of the original stack which stores the CZ assures the immutability of the stack for a given Element.

An Example of Using Conflict Zones

The following example is intended to clarify the operation of the mechanism for managing conflicting Resources. The example has been divided into two phases: first, note how the CZs are merged when two Elements are competing

for the same Resource; then, note the usefulness of the synchronisation stack as the two CZs whose stack is already in use are merged. Figure 4.5 shows the simulation components that intervene in the example.¹

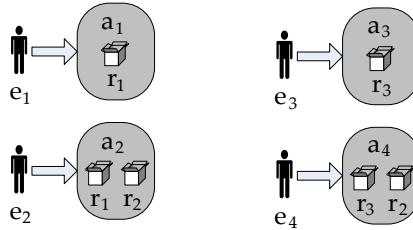


Figure 4.5. Example of several elements requesting activities in different AMs



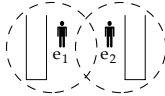
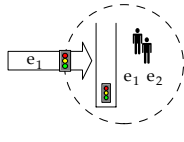
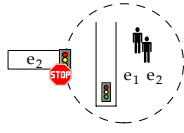
The first part of the example is shown in Table 4.2, which represents the concurrent execution of Elements e_1 and e_2 . The “execution” column indicates the stage each Element is in. The “conflict zone” column shows how the components in the CZ are modified. Finally, the “book list” column represents the progress in the Resource booking lists.

Table 4.2 shows how e_1 and e_2 initially have different CZs, containing only the corresponding Element and their own empty synchronisation stack. At stage 2, a conflict is detected and both CZs are merged. Now, at stage 3, e_1 and e_2 share the same CZ. Thus, when e_1 is in the *capture* phase, a synchronisation mechanism appears (represented by a traffic light) to properly arbitrate the exclusive access to the shared Resources. Consequently, when e_2 reaches the same phase (stage 4), the synchronisation mechanism prohibits this Element from accessing the Resources (represented as a STOP sign).

At this point, e_1 and e_2 share a CZ that is managed with one stack containing a single mechanism. The second part of the example includes Elements e_3 and e_4 , whose execution is analogous to that of e_1 and e_2 up to step 3. Table 4.3 shows the execution of e_3 and e_4 from that point on, and continues the example above.

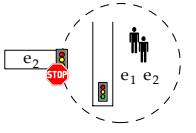
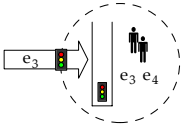
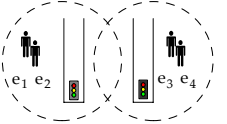
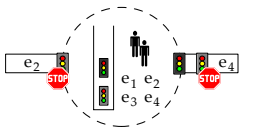
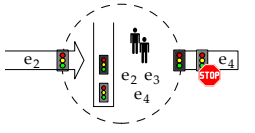
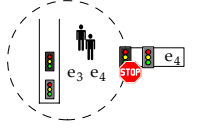
¹Note that Workgroups and even Resource Types are obviated for the sake of simplicity.

Table 4.2. Simple example of the resource booking phase

#	Execution		Conflict Zone		Book list	
	e_1	e_2	$CZ(e_1)$	$CZ(e_2)$	r_1	r_2
1	Init	Init			\emptyset	\emptyset
2	Book r_1	Book r_1	Merge($CZ(e_1), CZ(e_2)$) 		e_1	\emptyset
3	Capture				e_1, e_2	\emptyset
4		Book r_2			e_1, e_2	e_2
5		Capture			e_1, e_2	e_2

At stage 5, e_3 is given exclusive access to the CZ that it shares with e_4 . For this purpose, a new synchronisation mechanism (this time represented as a darker traffic light) is added. When e_4 books r_2 , the CZ for e_1 and e_2 has to be merged with that for e_3 and e_4 . Now, at stage 7, e_1 and e_3 can continue accessing the Resources since they do not really share any Resource. However, e_4 has to wait for e_3 and e_1 to finish before it can gain access to the shared Resources. This situation is represented by having both synchronisation mechanisms impede the progress of e_4 . Once e_1 finishes, e_2 can gain access to the CZ, but e_4 will wait until both e_2 and e_3 finish.

Table 4.3. Advanced example of the resource booking phase

#	Execution				Conflict Zone	
	e_1	e_2	e_3	e_4	$CZ(e_1, e_2)$	$CZ(e_3, e_4)$
5						
6				Book r_2	Merge($CZ(e_1, e_2), CZ(e_3, e_4)$) 	
7				Capture		
8	End					
9		End				

4.4. A Test Benchmark for SIGHOS

The following sections explain the different techniques applied to achieve an efficient and effective parallel version of SIGHOS. A test benchmark is required to compare the influence of the successive enhancements on the performance of the tool. The base test model is defined as follows:

- Let E be the set of Elements in the model. Let $|E|$ be the cardinal of E . All

the Elements are created at the beginning of the simulation.

- Let A be the set of Activities in the model, and let $|A| = k * |E|, k \in \mathbb{Z}^+$ be the cardinal of A .
- Each Activity $a_j \in A$ has a unique Workgroup $WG_j(\{\langle rt_1, 1 \rangle, \langle rt_2, 1 \rangle, \dots, \langle rt_\alpha, 1 \rangle\})$, with each Resource Type being used solely in a single Workgroup, that is, $\forall rt_i \in WG_j \Rightarrow rt_i \notin WG_k, k \neq j$. Hence, $|RT| = \alpha * |A|$.
- All the Activities have the same length.
- There are as many Resources R as Resource Types, that is $|R| = \alpha * |A|$.
- The Workflow associated with Element $e_i \in E$ involves iteratively requesting N times the Activity $a_j \in A$.

This simple test is intended to generate a large amount of simultaneous events at any simulation timestamp. Having as many Resources as Resource Types, all the Elements are always able to perform their requested Activities. We use this simple test to study how the simulator responds to several stress conditions by modifying certain input parameters ($|E|, N, |A|, \alpha, \beta$):

1. $|E|$ denotes the number of simultaneous events at any simulation timestamp.
2. N is a scale parameter which reduces the noise produced by other programs when measuring execution time (exclusive access to the processors during the execution cannot be granted).
3. $|A|$ defines the independence among events. Hence, if ($|A| = |E|$), all the Elements would request a different Activity, and all the events would be independent; in contrast, if ($|A| < |E|$), several Elements would request the same Activity, thus requiring mutual exclusion and creating dependencies among events.
4. α can be used to increase the workload of the request events.

5. This simple model can be made more complex by adding an extra parameter $\beta \in \mathbb{Z}^+ \leq |A|$. β would denote the number of multiple roles for which a Resource is potentially available. If $\beta = 1$ we would have a model equivalent to the simple model presented up to now; however, if $\beta > 1$, the same Resource r_k would be available for several Resource Types ($rt_1, rt_2, \dots, rt_\beta$). The Resource Types are chosen so that $\forall rt_i, 1 \leq i \leq \beta \Rightarrow \exists rt_j, 1 \leq j \leq \beta \wedge j \neq i / rt_i \in AM_k \wedge rt_j \in AM_k$. Hence, the mechanisms introduced in Subsection 4.3.3 would be put to the test.

All the experiments were executed on the platform described in Table 4.4.

Table 4.4. Test platform

Processors	16 (4 Quad Core AMD Opteron (2.2 GHz))
RAM	32 GB
OS	Linux 64 bits
JVM	1.6.0_11

4.5. A Performance Analysis of the Sequential SIGHOS

Now that a sufficiently flexible test benchmark has been established, let us analyse the performance of this model for the sequential version of SIGHOS on the platform described above. Table 4.5 enumerates the parameters used to characterise the different scenarios for the experiments.

We have selected large values for N to prevent external programs from influencing the measurements. The smaller value (10,000 iterations) can be used to check whether shorter simulations yield sufficiently accurate results. For example, Table 4.6 compares the execution time for the 10,000 and 100,000 iteration version by combining all the parameters for $\alpha = 8$. The “Rate” column is the quotient of the 100,000 and 10,000 iteration versions, which is, as expected, very close to 10 in most cases. Similar results are obtained for $\alpha = 4$ and $\alpha = 16$.

Table 4.5. Sequential parameters

Parameter	Values
N	10,000, 100,000
$ E $	512, 1024
$ A $	128, 512, 1024
α	4, 8, 16
β	1, 2, 4

Consequently, the execution times for the 10,000 iteration version are accurate enough to produce valid conclusions whenever required.

Table 4.6. 10,000 Vs 100,000 iterations ($\alpha = 8$)

β	$ A $	$ E $	Execution time (ms)		Rate
			10000 iter.	100000 iter.	
1	128	512	90,542	873,146	9.64
1	512	512	111,849	1,121,297	10.03
1	128	1024	195,057	1,950,204	10.00
1	512	1024	219,317	2,189,258	9.98
1	1024	1024	225,418	2,268,404	10.06
2	128	512	101,291	991,693	9.79
2	512	512	125,197	1,244,973	9.94
2	128	1024	225,737	2,259,927	10.01
2	512	1024	249,601	2,483,356	9.95
2	1024	1024	258,614	2,581,552	9.98
4	128	512	125,280	1,234,935	9.86
4	512	512	149,781	1,479,116	9.88
4	128	1024	287,201	2,860,710	9.96
4	512	1024	305,232	3,032,064	9.93
4	1024	1024	319,118	3,167,985	9.93

If treated independently, both $|E|$ and $|A|$ are scale factors, but their impact on the execution time is dramatically different, as shown in Figure 4.6. On the one hand, $|E|$ has a strong influence on the execution time, since its value is directly proportional to the number of events executed. On the other hand, when $|A|$ grows, the execution time only increases slightly.

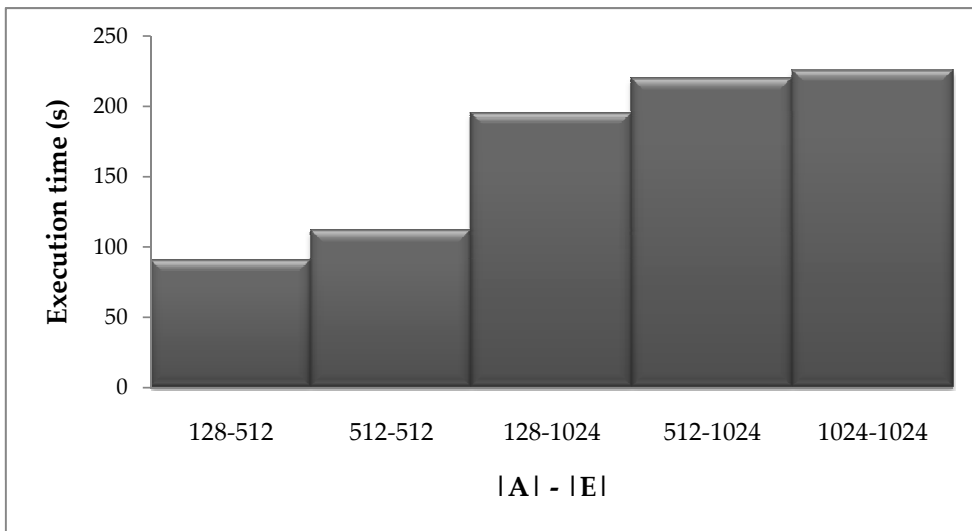


Figure 4.6. Comparison of influence of $|E|$ and $|A|$ for $\alpha = 8$ and $\beta = 1$

α and β also have a scale effect. Certainly, the sequential SIGHOS never uses the algorithm introduced in Subsection 4.3.3, thus increasing β simply means that more Resources are available for the same Resource Type. As seen in Figure 4.7 and Figure 4.8, α 's effect on execution time is almost linear, whereas β 's influence is much smoother.

In conclusion, SIGHOS behaves as expected and most parameters show a linear effect on the execution time.

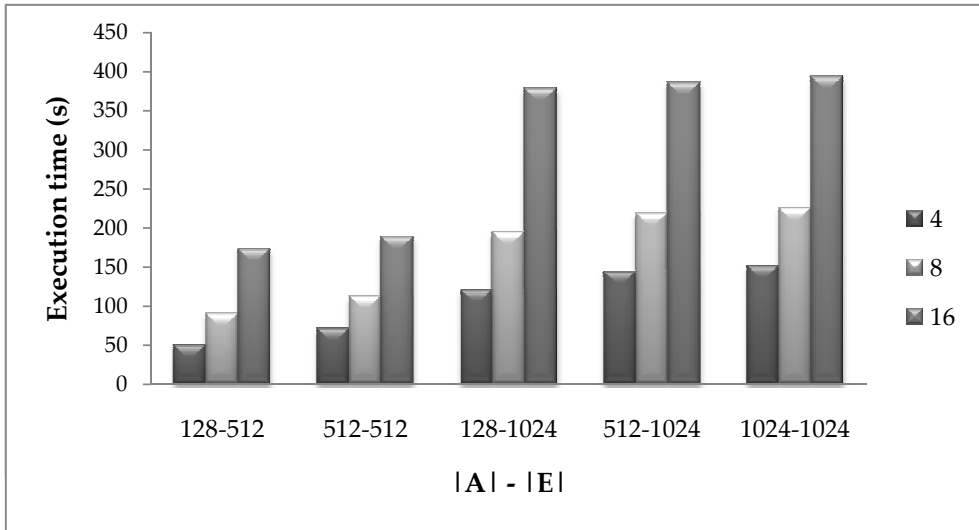


Figure 4.7. Effect of varying α for different problem types and $\beta = 1$

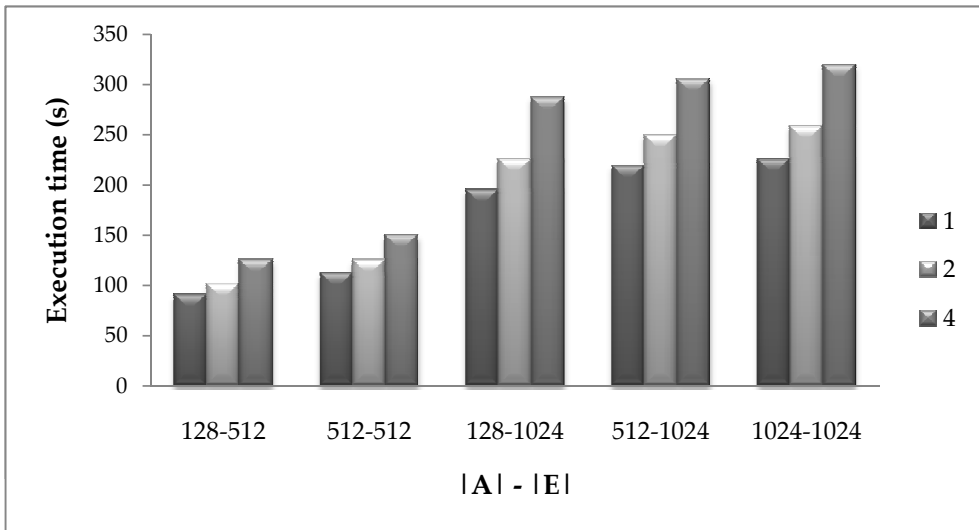


Figure 4.8. Effect of varying β for different problem types and $\alpha = 8$

4.6. Using External Event Executors

Having studied the performance of the sequential version of SIGHOS, we will now describe successive approaches for obtaining an efficient parallel version.

To start with, a direct mapping can be established between the concepts defined at the end of Section 4.1 and the usage of a thread pool (Goetz et al., 2006). A thread pool simply consists of a set of threads available to the application. Instead of creating a thread each time a task is to be performed, the task is placed in the pool manager's queue. As soon as one of the pool threads becomes available, it is assigned to this task.

The use of this pool in Java is as easy as choosing the `ExecutorService` class from the `java.util.concurrent` package, which can be parameterised with a wide variety of options for optimising the execution, depending on the problem's characteristics. A suitable starting point is defining the thread pool as having a fixed number of threads which are created when the executor is started.

```
ExecutorService tp = Executors.newFixedThreadPool(nThreads);
```

Obviously, EEs can be mapped to the threads of a thread pool. Further, the algorithm defined in Listing 4.1 can remain unaltered. The only requirement is to make `Events` extend the `Runnable` interface.

We can analyse the performance of the simulator when adding a thread pool by using the test benchmark previously defined. Nevertheless, the effects of some of the parameters in PSIGHOS are considerably different from those in SIGHOS.

1. Although $|E|$ and $|A|$ continue being scale factors, the quotient $|E|/|A|$ acquires a remarkable prominence in PSIGHOS. The closer the quotient is to 1, the higher degree of parallelism PSIGHOS may exploit. Consequently, contrarily to the sequential simulator, increasing $|A|$ should lead to better results when using several threads.
2. Now $\beta > 1$ involves using the algorithm explained in Subsection 4.3.3. A priori, this should result in poorer performance.

Hereafter, *speedup* is defined as the execution time of a sequential simulation performed with SIGHOS divided by the execution time of a parallel simulation performed with PSIGHOS. Let us select a scenario with $|E| = 512$ and $\alpha = 8$. Figure 4.9 shows the speedup obtained by using the thread pool with respect to the sequential simulator. At most 15 worker threads can be used, since the main thread is handling the future event list and sending events to be executed. None of the results are satisfactory except when 4 threads are used. Even in that case, when $|E|/|A| < 1$ the speedup is less than 1.

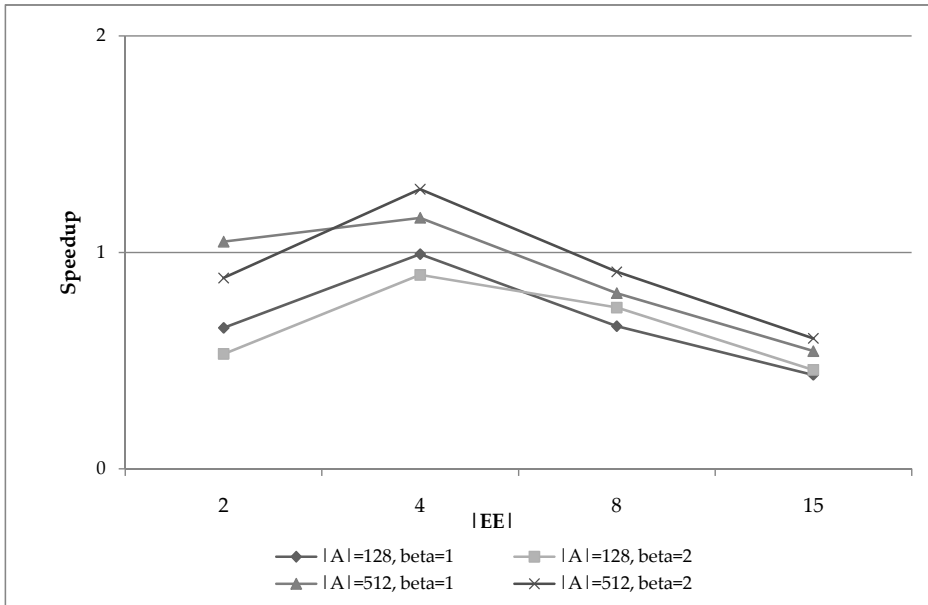


Figure 4.9. Speedup using Java thread pool with $|E| = 512$ and $\alpha = 8$

The main reason for this poor performance is the overload introduced to control the concurrent access to the simulator structures if compared to the small workload of the events. This effect can be observed if we lighten the workload by choosing $\alpha = 4$, as shown in Figure 4.10. In this case, none of the configurations achieves a speedup > 1 .

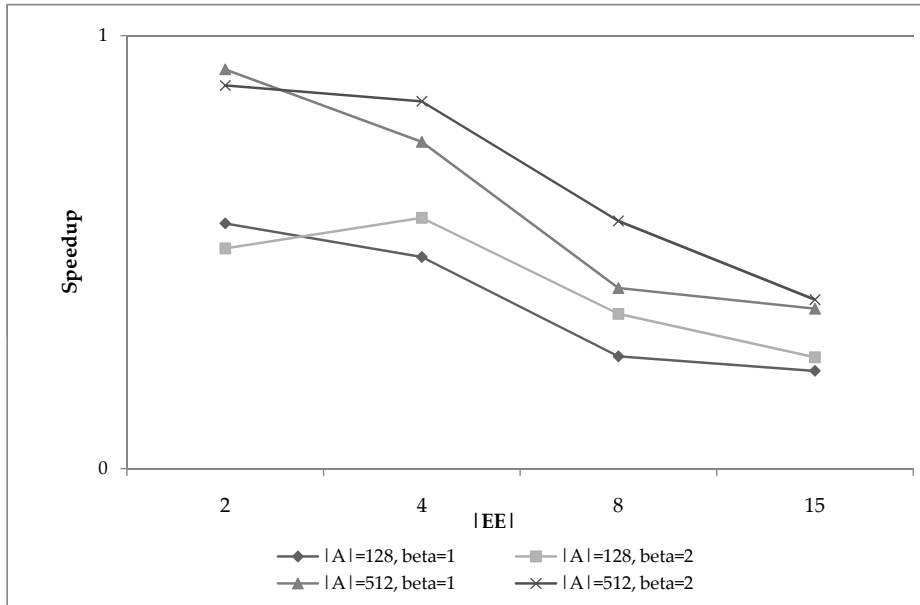


Figure 4.10. Speedup using Java thread pool with $|E| = 512$ and $\alpha = 4$

4.7. Integrating the Pool in the Simulation Tool

Java utilities for creating and handling a thread pool are simple and clean, and the development times are considerably reduced; nevertheless, an ad hoc solution may easily surpass the efficiency of a generic tool like this when applied to a specific problem. Therefore, an `EventExecutor` class is explicitly created within the library. Each EE is a thread whose basic behaviour is described in Listing 4.2.

This scheme allows for some optimisation if we take into consideration the fact that every event (save for those generated during the initialisation phase) is created by other events. Sending these events to the master thread to be distributed again is expensive, as opposed to keeping them in the same EE for execution. While this could certainly lead to an unbalanced load, the costs of migrating (small pieces of) work would be more harmful. The implementation of this op-

timisation involves adding an execution cache to the EE. This cache temporarily keeps the new events generated during the execution of the event sent by the EM. This can be done by modifying the `schedule(e@ts)` operation, and the main EE loop, as seen in Listing 4.3 and in Listing 4.4 respectively.

Listing 4.2. Basic definition of an ad-hoc Event Executor

```
EventExecutor {
    startEvent = Event used as an atomic flag;

    method dispatch(e@ts) {
        // Atomic CAS operation
        if (startEvent ≠ ∅) {
            startEvent = e@ts;
        }
    }

    method mainLoop() {
        while (Simulation.CST < Simulation.EndTs) {
            if (startEvent ≠ ∅) {
                execute(startEvent);
                startEvent = ∅;
            }
        }
    }
}
```

Listing 4.3. First optimisation of an ad-hoc Event Executor

```
method schedule(e@ts) {
    if (ts = Simulation.CST) {
        ee = Event Executor where the current event is being executed
        ee.executionCache.add(e@ts);
    }
    else if (ts > Simulation.CST) {
        Simulation.FEL.add(e@ts);
    }
}
```

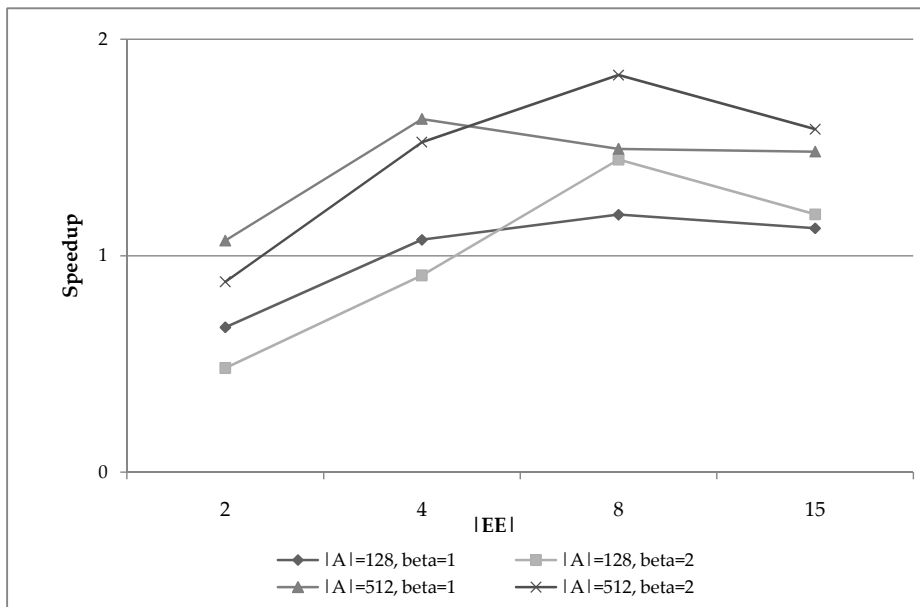
Listing 4.4. Modified main loop of an ad-hoc Event Executor

```

while (Simulation.CST < Simulation.EndTs) {
  if (startEvent  $\neq \emptyset$ ) {
    execute(startEvent);
    forall e@ts  $\in$  executionCache do {
      execute(e@ts);
    }
    startEvent =  $\emptyset$ ;
  }
}

```

Figure 4.11 illustrates the benefits of the ad hoc solution. The resulting speedup is still small, but the simulator's behaviour when the number of threads is increased is much more stable.

**Figure 4.11.** Speedup using an ad hoc thread pool with $|E| = 512$ and $\alpha = 8$

4.8. Exploiting Event Locality

One of the major drawbacks of the previous approach is the inefficiency of accessing the FEL, since both the EM and the EEs concurrently try to gain access to this structure. We solve this problem by modifying the `schedule(e@ts)` operation again, as shown in Listing 4.5.

Listing 4.5. The `schedule(e@ts)` operation modified to exploit event locality

```
method schedule(e@ts) {
    ee = Event Executor where the current event is being executed
    if (ts = Simulation.CST) {
        ee.executionCache.add(e@ts);
    }
    else if (ts > Simulation.CST) {
        ee.FELCache.add(e@ts);
    }
}
```

The new `schedule@ts` operation extends the optimisation introduced in the previous section to future events, that is, events with timestamps in the future may be stored in caches until all the events corresponding to the current timestamp are executed. The EM updates the FEL by checking those caches before advancing the VC. Thus, only the EM accesses the FEL and no mechanisms to protect that structure are needed. Figure 4.12 schematises the interaction among the EM, EEs and the simulation structures after applying the optimisation.

Figure 4.13 illustrates the results obtained with this optimisation. The most notable effect is a higher scalability, especially for $|EE| \geq 8$, where the speedup almost doubles the previous one.

4.9. Block Dispatching

In Section 4.2 we stated that sufficiently large and complex simulations have multiple events per timestamp. Starting from this hypothesis, the `dispatch(e@ts)` operation may be replaced by a `dispatch(E)` operation, where E is a set of events.

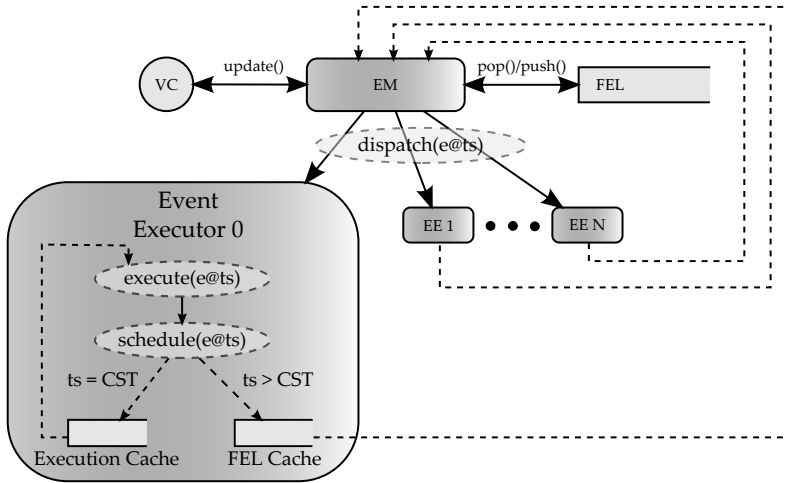


Figure 4.12. Basic schema of the optimised Master-Slave approach

Hence, events would be grouped and dispatched in *blocks*, as shown in Listing 4.6.

Listing 4.6. Event Manager main loop using block dispatching

```

Let executor[] be an array containing all the Event Executors
// Phase 0: Initialisation
...
while (Simulation.CST < Simulation.endTs) {
  // Phase 1: Update clock
  ...
  // Phase 2: Execution
  E =  $\forall e@ts \in \text{Simulation.FEL} / (ts = \text{Simulation.CST});$ 
  block = E.length / executor.length;
  for i  $\in [0 .. \text{executor.length} - 1]$  {
    E' = E[(i * block) .. ((i + 1) * block)];
    executor[i].dispatch(E');
  }
  // Synchronisation of Phase 2
  ...
}

```

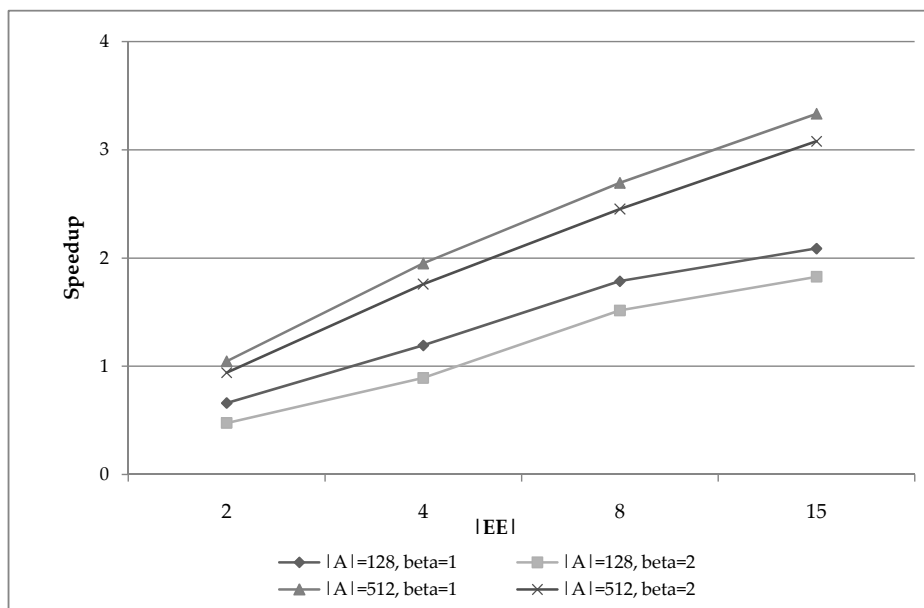


Figure 4.13. Speedup exploiting event locality with $|E| = 512$ and $\alpha = 8$

The practical implementation of this algorithm distinguishes between two special cases:

1. If there are fewer events than EEs, all the events are sent to the first executor since the workload is not worth the penalty imposed by the synchronization operations.
2. If $E.length \bmod executor.length \neq 0$, the remaining events are sent to the last EE.

Figure 4.14 shows the speedup in the simulator when using block dispatching in addition to the previous enhancements. The new results are a considerable improvement on those obtained earlier.

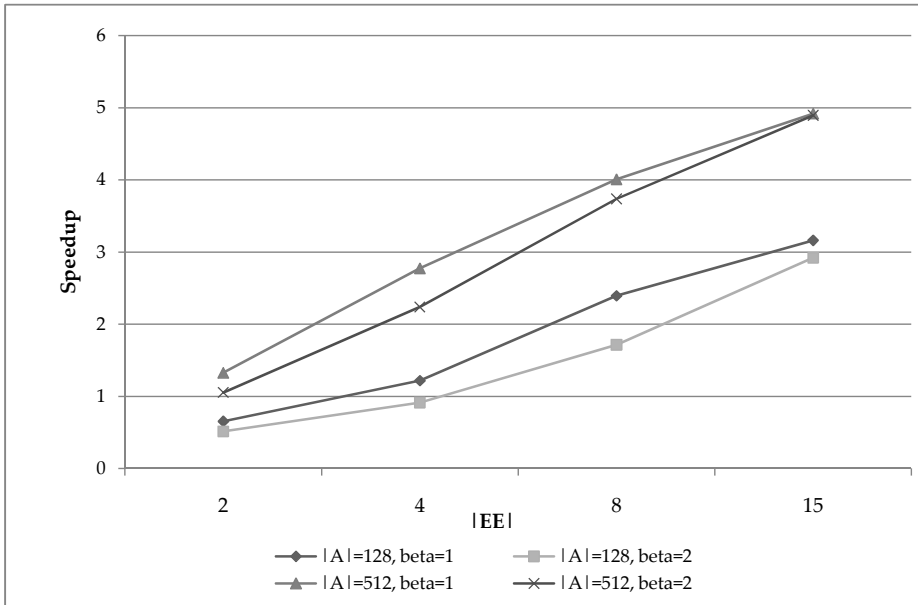


Figure 4.14. Speedup exploiting block dispatching with $|E| = 512$ and $\alpha = 8$

4.10. A Hybrid Event Manager - Executor

In a scenario like the one presented in the previous section, it is clear that the workload of the EM has been considerably reduced with respect to that of the EEs.

This observation presents an opportunity to improve the performance of the simulator, since the EM may play the role of an additional EE, and execute part of the events corresponding to the current simulation time. Moreover, there is no need for said EM to dispatch the events: each EE could simply be notified as soon as the VC is updated and access its corresponding event subset based on its index. Figure 4.15 schematises this new approach. A kind of “master” EE (the hybrid EM-EE, hereafter EME) is still required to perform the common tasks, but the original MS paradigm is considerably relaxed.

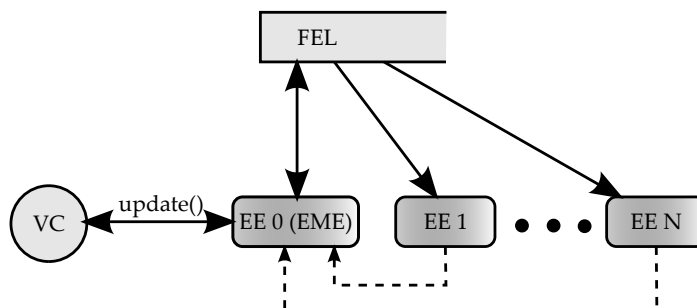


Figure 4.15. Relaxed MS approach by using the hybrid EME

Listing 4.7 presents the modified EE's main loop. EE with id = 0 is considered the hybrid Event Manager Executor (EME) and substitutes the former EM. Since the dispatch(e@ts) operation is no longer used, two operations, notify() and await(), are included for synchronisation purposes.

Listing 4.7. Main loop of an EE and the hybrid EME

```

while (Simulation.CST < Simulation.EndTs) {
    E =  $\forall$  e@ts  $\in$  Simulation.FEL / (ts = Simulation.CST);
    block = E.length / executor.length;
    executionCache = E[(id * block) .. ((id + 1) * block)];
    forall e@ts  $\in$  executionCache do {
        execute(e@ts);
    }
    // If this is the EME
    if (id == 0) {
        // Synchronisation
        Wait until all the events has been effectively executed
        // Update FEL
        for i  $\in$  [1 .. executor.length - 1] {
            FEL.addAll(executor[i].FELCache);
        }
        // Update clock
        e'@ts = First event from FEL;
        Simulation.CST = ts;
    }
}

```

```

    for i ∈ [1 .. executor.length - 1] {
        executor[i].notify();
    }
}
// If this is a "normal" EE waits until the EME notifies
else {
    await();
}
}

```

Unfortunately, the experimental results do not support the previous assumption. Figure 4.16 compares the speedup obtained when the EM dispatches and when it dispatched and executes events. Note how the former outperforms the latter. The last column compares the maximal configuration for each version: even with an additional EE, the hybrid EME cannot reach the speedup obtained with the previous version.

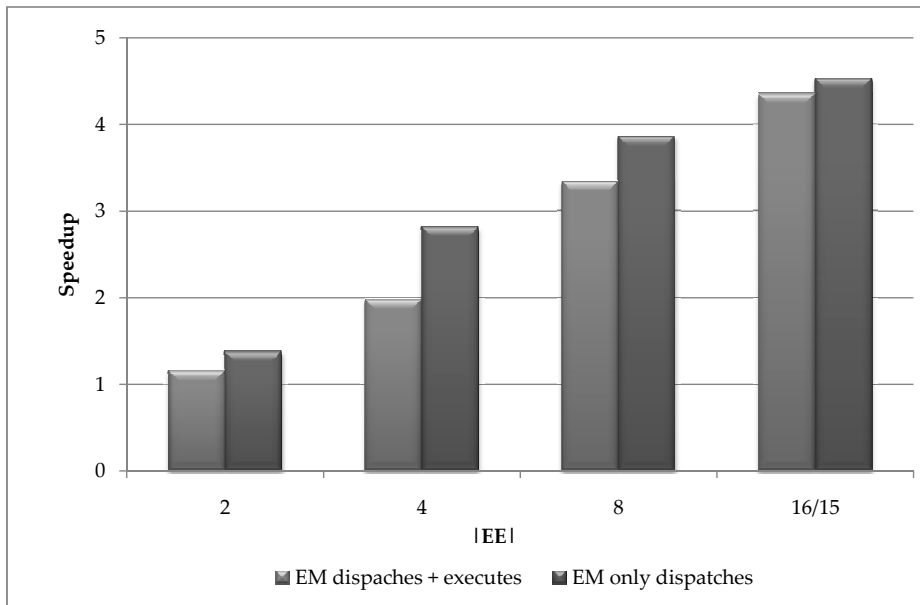


Figure 4.16. Speedup compared with $|E| = |A| = 512$, $\alpha = 4$ and $\beta = 1$

This counterintuitive result leads us to more carefully analyse the workload of the EME compared to that of the EEs. Hence, we design a mechanism to control how many events are dispatched to each worker based on two parameters: *grain* and *rest*. The events that must be executed are organised into b bundles, where $b = |EE| * grain + rest$. Let λ be the number of events to be dispatched. Each EE executes $\lfloor \lambda/b \rfloor * grain$ events, whereas the EME executes the remaining $\lambda - \lfloor \lambda/b \rfloor * grain * |EE|$ events. Figure 4.17 compares the performance for 6 different configurations: the first one (*Simple EM*) uses the original simulation engine with an EM exclusively devoted to dispatching events; the remaining configurations set different (*grain*, *rest*) parameter values for the EME engine. Note that, this time, the chart is comparing the results with respect to the number of threads, not only the EEs. Hence, the 2-thread bar for the *Simple EM* configuration has a single EE.

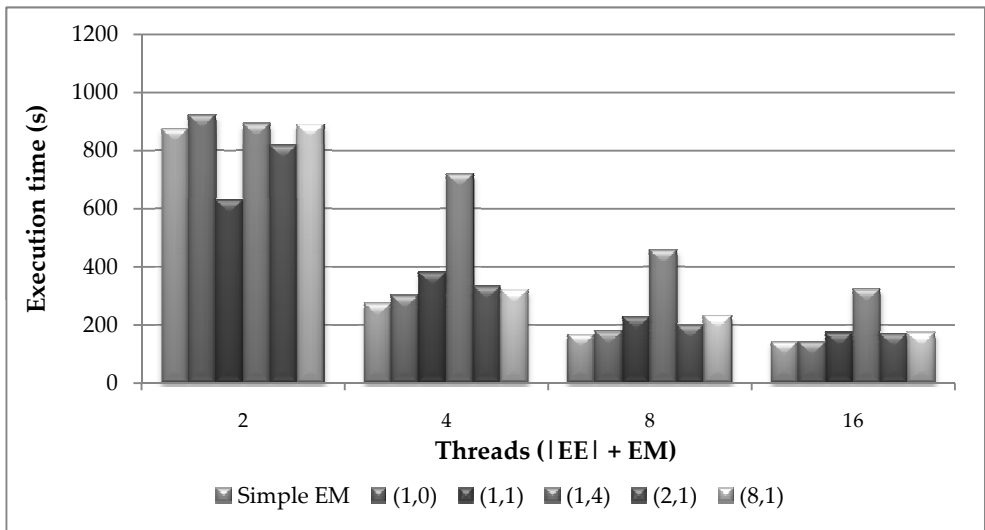


Figure 4.17. Performance comparison with $|E| = |A| = 512$, $\alpha = 4$ and $\beta = 1$ for different *grain-rest* configurations

The different configurations tested shed light on the unexpected behaviour of the hybrid EME simulation engine. Hence, the $(1,0)$ ($grain = 1$ and $rest = 0$) configuration serves to validate the *Simple EM* configuration, since it involves an EM that does not execute events.

The strictly fair distribution of events $(1,1)$ even outperforms the *Simple EM* case when only two threads are available. Obviously, the $(1,1)$ case effectively has two threads for executing events, whereas only one thread from the *Simple EM* configuration is an EE. However, when more than two threads are involved, this good behaviour disappears.

The EME is assigned a considerably higher workload than the EEs when $(1,4)$ is chosen. All but the 2-thread scenario offer the worst performance, thus corroborating the fact that a high workload in the EME is counterproductive.

Finally, the EEs are assigned twice/eight times the workload of the EME ($(2,1) / (8,1)$). None of these configurations achieves better performance than either the *Simple EM* or the $(1,0)$ configuration. Moreover, some other different *grain* values were tested (4, 6...) with similar results. This suggests that lowering the workload of the EME does not reduce the execution time. In conclusion, what penalises the performance is not the execution of events, but the EME tasks involving dispatching events and synchronising the simulation.

4.11. Going Beyond Limits: like-3-Phase Approach

Caches and block dispatching are techniques that considerably improve the performance of PSIGHOS. Nevertheless, the results for the best scenarios in our test benchmark ($|A| = |E|$ and $\beta = 1$) are far from satisfactory, especially in terms of the poor scalability as the number of threads increases.

The source of the problem lies in an excessive amount of protected simulation components: *Elements*, *Resources*, *ActivityManagers*... require controlling the access to their state, since several events may be concurrently trying to modify them.

A coarse grain access control is exerted on these structures, since we cannot predict which events will execute concurrently or what simulation components they will affect.

A second problem is the redundancy of actions. Consider a scenario with a single Activity a_1 that requires one of each Resource (rt_1, rt_2, rt_3). Hence, only AM_1 is needed to manage the accesses to a_i , rt_1 , rt_2 and rt_3 . Currently ($CST = t$), there are no available Resources, but an Element e_1 is already waiting in a_1 's queue. Assume there is one Resource per RT (r_1, r_2, r_3), and all three Resources becoming available, that is, scheduling a RoleOn event, at timestamp $t + 1$. When the VC reaches $t + 1$, those three events are executed. A RoleOn event notifies the corresponding AM that a new Resource is available, and checks if any of the Activities $\in AM$ with non-empty Element queues are now feasible. Suppose that the events are executed in the order shown in Table 4.7. Then, $r_1.RoleOn@(t + 1)$ is the first event to launch the check routine, preventing the control mechanism of AM_1 from executing this action simultaneously with any other event. $r_1.RoleOn@(t + 1)$ will check if a_1 is feasible and will fail (since we still require at least one Resource for rt_2 and rt_3). The same action and the same result will occur with the second event. The third event will succeed, but two events have been wasted because the simulation engine cannot know in advance that more RoleOn events affecting the same AM will occur. Furthermore, if we analyse the costs, in terms of concurrent access, related to executing each of these unnecessary events, we realise that exclusive access to AM_1 , e_1 , and each of the resources is required, thus limiting other opportunities to exploit parallelism with these simulation components.

Table 4.7. Three consecutive RoleOn events with the same timestamp

Event	rt_1	rt_2	rt_3	Is a_1 feasible?
$r_1.RoleOn$	✓	✗	✗	✗
$r_2.RoleOn$	✓	✓	✗	✗
$r_3.RoleOn$	✓	✓	✓	✓

4.11.1. Events Revisited

From the example above, we deduce that the order in which the events are executed and, more specifically, the knowledge regarding how many events are left that affect the same simulation component, have a dramatic influence on the performance of the simulation engine. A careful assessment of the operations performed within SIGHOS events, and the simulation components they affect, may lead to a reformulation of the events themselves. Hence, a review of the event code is required. Let `acquire()` and `release()` be operations defined within any simulation object (`AM`, `Element`, `Resource...`) to acquire and release exclusive access to such object, respectively. The entire code between `object.acquire()` and `object.release()` is executed atomically from the point of view of `object`. Listings 4.8, 4.9 and 4.10 complete the pseudocode already shown in listings 3.1, 3.2 and 3.3 with the corresponding acquire/release operations.

Listing 4.8. RequestActivity event

```
RequestActivity(Element e, Activity a) {
    // a ∈ am
    am.acquire();
    e.acquire();
    if (a.isFeasible(e)) { // May involve invocations to r
        .acquire/release
        e.release();
        e.carryOut(a); // May involve invocations to r.acquire/release
        Simulation.schedule(new FinalizeActivity(e, a, Simulation.CST + a
            .duration));
    }
    else {
        e.release();
        a.enqueue(e);
    }
    am.release();
}
```

Listing 4.9. FinalizeActivity event

```
FinalizeActivity(Element e, Activity a) {
    // a.releaseResources() is inlined
    AMSet =  $\emptyset$ ;
    forall  $r_i \in$  (list of resources captured by e to carry out a) do {
         $r_i$ .AMSet = set of AMs where  $r_i$  is currently available
        AMSet = AMSet  $\cup$   $r_i$ .AMSet;
    }
    forall  $am_i \in$  AMSet do {
         $am_i$ .acquire();
         $am_i$ .notifyAvailableResource();
         $am_i$ .release();
    }
    // The rest of the pseudocode is kept the same
    forall  $a_i \in$  (list of activities currently requested by e) do {
        Simulation.schedule(new AvailableElement(e,  $a_i$ , Simulation.CST));
    }
    // Make e advance its associated workflow
    forall  $a_j \in$  (next activities to be requested according to the
        associated workflow) do {
        Simulation.schedule(new RequestActivity(e,  $a_j$ , Simulation.CST));
    }
}
```

Listing 4.10. AvailableElement event

```
AvailableElement(Element e, Activity a) {
    //  $a \in am$ 
    am.acquire();
    e.acquire();
    if (a.isFeasible(e)) { // May involve invocations to r
        .acquire/release
        e.release();
        e.carryOut(a); // May involve invocations to r.acquire/release
        Simulation.schedule(new FinalizeActivity(e, a, Simulation.CST + a
            .duration));
        a.dequeue(e);
    }
}
```



```
    else {  
        e.release();  
    }  
    am.release();  
}
```

The contents of the `RoleOn` and `RoleOff` events are also displayed in listings 4.11 and 4.12. These events simply increment/decrement the counter of currently available `Resources`; update the `Resource` state to take into account that it is available/unavailable for certain roles; and notify the `AM` (in the case of `RoleOn`) that a new resource is available.

Listing 4.11. `RoleOn` event

```
RoleOn(Resource r, TimeTableEntry tte) {  
    // tte.rt ∈ am  
    am.acquire();  
    tte.rt.incAvailable(r);  
    r.acquire();  
    r.AMSet.add(am); // AMSet is the set of AMs where r is available  
    r.release();  
    am.notifyAvailableResource();  
    am.release();  
    Simulation.schedule(new RoleOff(r, tte))  
}
```

Listing 4.12. `RoleOff` event

```
RoleOff(Resource r, TimeTableEntry tte) {  
    // tte.rt ∈ am  
    am.acquire();  
    tte.rt.decAvailable(r);  
    r.acquire();  
    r.AMSet.remove(am); // AMSet is the set of AMs where r is available  
    r.release();  
    am.release();  
    if (tte.c.hasNext())  
        Simulation.schedule(new RoleOn(r, tte))  
}
```

The `AM.notifyAvailableResource` method appears in both `RoleOn` and `FinalizeActivity`. Listing 4.13 shows the pseudocode corresponding to that method.

Listing 4.13. `AM.notifyAvailableResource` method

```
method notifyAvailableResource() {
  forall  $a_i \in$  current AM do {
    forall  $e_j \in a_i.queue$  do {
      e.acquire();
      if (a.isFeasible(e)) { // May involve invocations to r
        .acquire/release
        e.release();
        e.carryOut(a); // May involve invocations to r
          .acquire/release
        Simulation.schedule(new FinalizeActivity(e, a,
          Simulation.CST + a.duration));
        a.dequeue(e);
      }
      else {
        e.release();
      }
    }
  }
}
```

From the listings above we can conclude that, although the exclusive access to both `Elements` and `Resources` does not appear very often and is reasonably localised, the EEs spend most of the time accessing AMs in mutual exclusion (hereafter `mutex`). Furthermore, within the code inside the AM's `mutex` area, a pair of operations appears very frequently: `a.isFeasible(e)` and `e.carryOut(a)`. These operations are part of the two-phase acquisition of resources required to carry out an activity (checking if an activity is feasible with `a.isFeasible(e)`, and then effectively acquiring the corresponding resources with `e.carryOut(a)`), and comprise a significant portion of the computational load in SIGHOS. Therefore, `mutex` access to AMs represents a good starting point for optimising PSIGHOS.

4.11.2. Reducing Contention in AMs

In spite of being a suitable mechanism for preventing resource contention and deadlocks, AMs seem to be limiting the amount of parallelism that may be exploited in PSIGHOS. After all, the problem continues to be that the simulation engine has to handle many concurrent events that affect the portion of the system state protected by the AM.

Suppose we can assure that an AM will be always accessed from the same EE. In this case, we could avoid the AM's acquire/release operations. Unfortunately, there is no direct map *Event-AM*, since some events involve accessing more than one AM. Further, some events whose main action involves one AM may generate new events to be executed immediately on a different AM. For example, consider an Element performing a workflow consisting of two consecutive Activities a_1 and a_2 ($a_1 \in AM_1 \wedge a_2 \in AM_2$). Immediately after a `FinalizeEvent@ts` is invoked on a_1 , a `RequestEvent@ts` on a_2 is called.

Figure 4.18 schematises the order in which events are invoked. A distinction is made between two different invocation types: events that are scheduled for future execution (after Δt); and events that are scheduled for immediate execution, generally if a certain condition holds. This distinction clearly meets the definitions of the *B* and *C* events in the three-phase approach (Subsection 1.2.4). Hence, `Generate`, `FinalizeActivity`, `RoleOn` and `RoleOff` would be *B* events; whereas `RequestActivity`, `AvailableElement` would be *Cs*². Nevertheless, this rough assignment is useless if the events are not more precisely decomposed.

Let us focus on the operations performed within the mutex area of the AMs. More specifically, let us concentrate on the pair `a.isFeasible(e) - e.carryOut(a)`. There are two triggers that may result in those operations being invoked:

1. One or more resources become available (`RoleOn` and `FinalizeActivity`) events.
2. An element becomes available (`AvailableElement` event).

²The "Start..." events may also be considered *Cs*, but we will exclude them from this discussion for the sake of simplicity

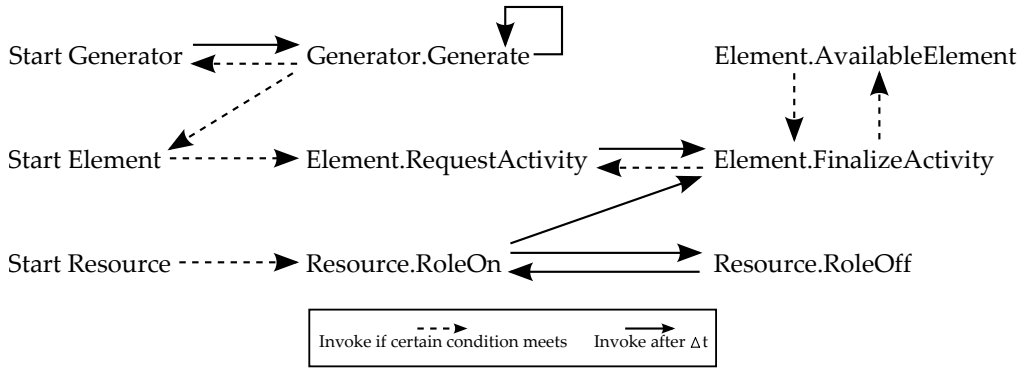


Figure 4.18. Relationships among events

The first case requires checking all the Activities belonging to an AM, as seen in `notifyAvailableResource`; whereas the second one can be reduced to a check of whether the Element that just became available can now perform any of the Activities it was waiting for. A few conclusions can be drawn from these premises:

- Case 1 should be executed only once per simulation clock.
- Case 1 includes case 2.
- Case 2 focuses on specific elements and specific activities.

We can implement cases 1 and 2 as a large like-C event attached to an AM, as can be seen in Listing 4.14. Case 1 simply invokes `AM.notifyAvailableResource`. The condition for launching this event may be a simple boolean flag (`flagRes`), activated if *at least one resource has become available for a resource type belonging to this AM*. The main condition for case 2 is that the code for case 1 was not already executed, and requires a list of pairs $\langle e_i, a_j \rangle$ such that e_i has become available $\wedge a_j$ was requested (and not yet carried out) by e_i . Let `requestList` be the name of that list.

Listing 4.14. AM event

```

AMEvent(am) {
  if flagRes = true {
    am.notifyAvailableResource();
  }
  else {
    forall pairs  $\langle e_i, a_j \rangle \in \text{requestList}$  do {
      e.acquire();
      if (a.isFeasible(e)) { // May involve invocations to r
        .acquire/release
        e.release();
        e.carryOut(a); // May involve invocations to r
          .acquire/release
        Simulation.schedule(new FinalizeActivity(e, a,
          Simulation.CST + a.duration));
        a.dequeue(e);
      }
      else {
        e.release();
      }
    }
  }
}

```

This large event not only avoids any kind of redundancy, but it prevents the AM from requiring a coarse-grain mutex access. Only `flagRes` and `requestList` demand a fine-grain protected access. Each `AMEvent` is assigned to a single EE and executed after the other regular events have finished, as seen in Listing 4.15 (which also includes some modifications due to the block dispatching). Actually, regular events are reduced to `Generate, RoleOff; RoleOn`, which only sets the corresponding `flagRes` to true; and `FinalizeActivity`, which only adds a new pair $\langle e_i, a_j \rangle$ to the corresponding AMs.

Listing 4.15. Modified main loop of a like-3-phase Event Executor

```
startEvents =  $\emptyset$ ; // Empty set of events
while (Simulation.CST < Simulation.EndTs) {
    if (startEvents  $\neq \emptyset$ ) {
        executionCache = startEvents;
        forall e@ts  $\in$  executionCache do {
            execute(e@ts);
        }
        startEvents =  $\emptyset$ ;
    }
    // Synchronisation of Phase 2
    Wait until all the events have been effectively executed
    // Phase 3: AM Conditional execution
    forall ami  $\in$  (set of assigned AMs) do {
        execute(ami.Event@ts);
    }
}
```

The assignment of AMs is performed during the initialisation phase of the EM's main loop, as seen in Listing 4.16. Whereas EEs have to synchronise during the second phase by detecting that all the regular events have been executed, the EM's main loop sets a second synchronisation after all the AM events have been completed.

Listing 4.16. Event Manager main loop using a like-3-phase approach

```
Let executor[] be an array containing all the Event Executors
// Phase 0: Initialisation
...
forall ami  $\in$  Simulation.AM do {
    executor[i % executor.length].assign(ami);
}
while (Simulation.CST < Simulation.endTs) {
    // Phase 1: Update clock
    ...
    // Phase 2: Execution
```

```

E =  $\forall$  e@ts  $\in$  Simulation.FEL / (ts = Simulation.CST);
block = E.length / executor.length;
for i  $\in$  [0 .. executor.length - 1] {
    E' = E[(i * block) .. ((i + 1) * block)];
    executor[i].dispatch(E');
}
// Synchronisation of Phase 3
Wait until all the AM events have been executed;

```

Figure 4.19 shows the speedup with this new approach, and Figure 4.20 compares the results of this approach with the block dispatching method.

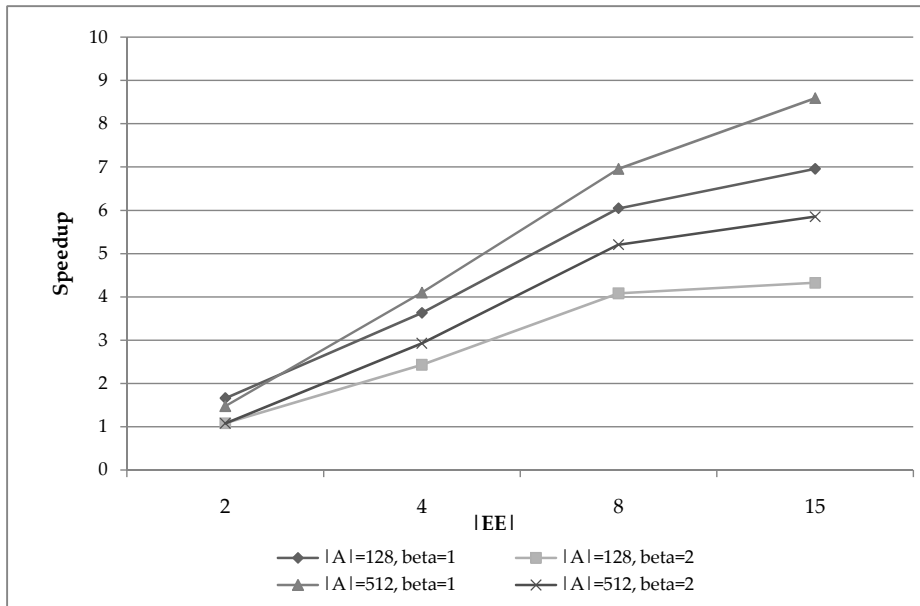


Figure 4.19. Speedup for the like-3-phase approach with $|E| = 512$ and $\alpha = 8$

A considerable improvement is obtained, especially noticeable for $|EE| = 8$, where the new speedup almost doubles the previous one. The result for $|EE| = 4$ requires more attention since it appears to be superlinear. There are several reasons that justify this value. First, as mentioned earlier, the like-3-phase

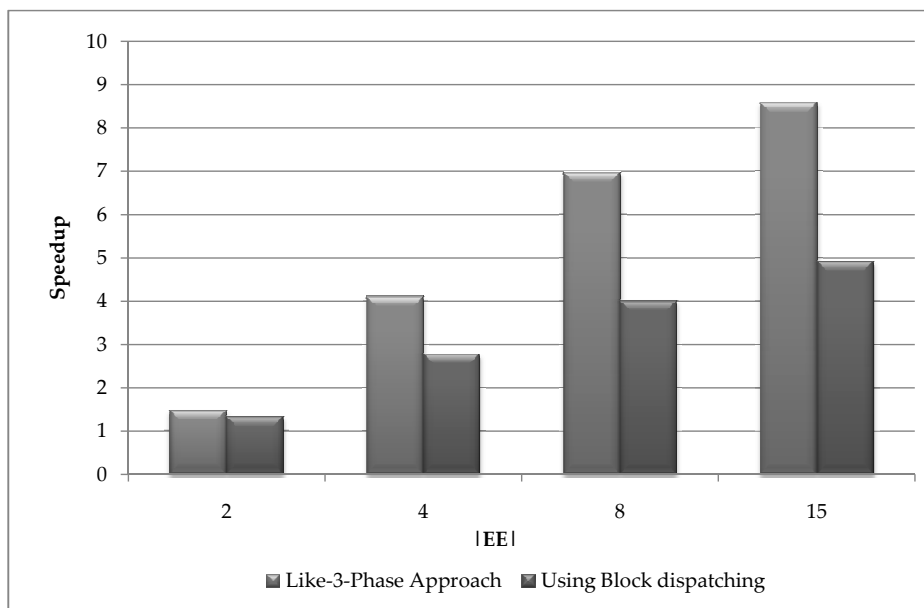


Figure 4.20. Comparison between the like-3-phase approach and the block dispatching method with $|E| = |A| = 512$, $\alpha = 8$ and $\beta = 1$

algorithm avoids many redundant operations that must be performed in the original algorithm. Second, the more processors, the more the cache available to carry out the simulation. Finally, even when the speedup may be considered to be superlinear with respect to $|EE|$, it must be noted that, in reality, $|EE| + 1$ processors are being used due to the EM. Hence, we cannot properly talk about superlinear speedup in this case.

4.12. Some Final Notes about the Implementation

Java and, more specifically, Java for concurrent programming, exhibits unpredictable behaviour that strongly affects the development of an efficient application. In this section we will discuss some subtle problems involving the practical

implementation of the solutions presented in this chapter. The conclusions derived from this chapter will be used for subsequent enhancements to PSIGHOS.

4.12.1. Spinlock vs Semaphore

The modifications introduced in Section 4.10 did not serve to achieve better results, but they do offer a suitable framework for discussing an implementation detail that strongly affects the performance of an application. When applying this approach, the EM is replaced by a hybrid EME that updates the VC and notifies the remaining EEs that new events are available for execution. An `await()` operation implements the wait for new EEs events. This section will show that the practical implementation of such an operation can considerably affect performance. Specifically, two implementations will be compared: using a Semaphore, like the one included in the `java.util.concurrent` package; and using an atomic variable in a spinlock.

Listing 4.17 describes the corresponding `await()` and `notify()` methods for the *Semaphore* approach.

Listing 4.17. `await()` method with `java.util.concurrent.Semaphore`

```
Semaphore lock = new Semaphore(0);
void notify () {
    lock.release();
}

void await () {
    try {
        lock.acquire();
    } catch (InterruptedException e) {
        . . .
    }
}
```

The second implementation is supported by the classes for implementing atomic variables provided by the `java.util.concurrent.atomic`. Listing 4.18 shows an example using an `AtomicBoolean`.

Listing 4.18. `await()` method with `java.util.concurrent.atomic.AtomicBoolean`

```
AtomicBoolean flag = new AtomicBoolean(false);

void notify() {
    flag.set(true);
}

void await() {
    // Spinlock
    while (!flag.compareAndSet(true, false));
}
```

On the surface, the semaphore solution seems more elegant and Java sound. However, Figure 4.21 offers some relevant results on the advantages of using the second approach over the first one. The differences are minimal for a reduced number of EEs (2-4), but a gain of around 40% is obtained for 15-16 EEs.

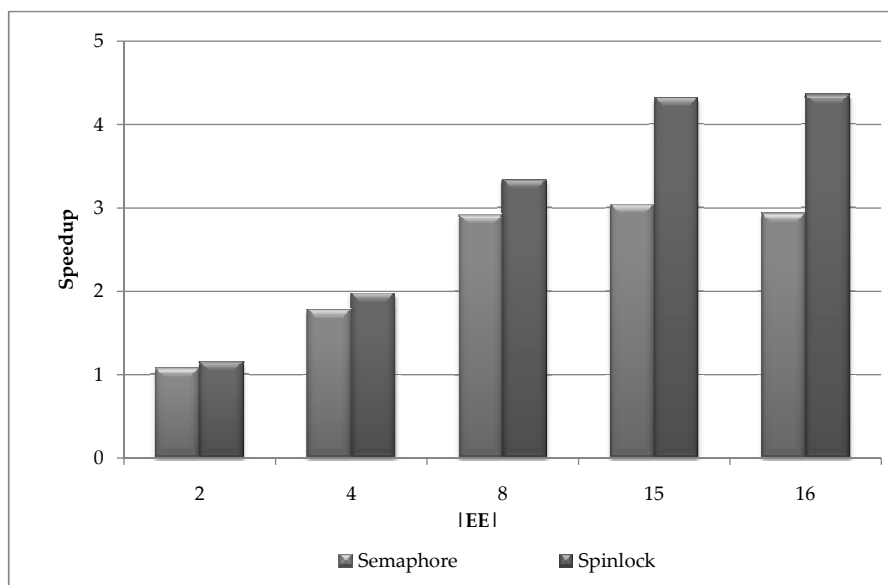


Figure 4.21. Comparison between using `java.util.concurrent.Semaphore` and a spinlock with a `java.util.concurrent.atomic.AtomicBoolean` with $|E| = |A| = 512$, $\alpha = 4$ and $\beta = 1$

The semaphore strategy for blocking/unblocking a thread relies on the invocation of `LockSupport.park()/LockSupport.unpark()` (Lea, 2005), that is, the thread is suspended and later resumed. This strategy is generally preferred to using a spinlock, since it wastes fewer CPU cycles, precludes starvation and scales better when the number of threads is higher than the number of physical computer processors. Nevertheless, there are two main reasons that make spinlock more appealing for our problem:

1. Using more threads than physical processors makes no sense in a tool that is supposed to perform extensive computations.
2. The objective is to maximise the performance of the simulator, even if the throughput of the computer on which the simulator is running is reduced. In other words, wasting a few CPU cycles is worth it if the general performance of the simulator is improved.

4.12.2. Thread vs Runnable

Java reference books describe two ways to create a thread: extending the `Thread` class, and implementing the `Runnable` interface. Programmers generally use the former or the latter according to their specific needs and the semantics they want to express. Nevertheless, due to the limited inheritance power of Java, the preferred approach is usually to implement the `Runnable` interface, just in case the new class is going to be inserted into an existing class hierarchy.

Performance is neglected as a factor to consider when choosing one or the other alternative. However, as we will prove, the simulation execution time can vary considerably depending on the method selected.

As seen in Section 4.6, `SIGHOS` events implement the `Runnable` interface since Java classes for a thread pool require so. While there was no need to keep this interface once the thread pool disappeared from the `SIGHOS` design, it was maintained for backward compatibility. Although `Runnable` is implemented, events are never *started* as threads, but rather are used as pieces of work or tasks. Ne-

vertheless, there is no reason (apart from good practice) not to extend Thread instead of implementing Runnable. Surprisingly, the results shown in Figure 4.22 prove that extending Thread strongly penalises the execution time.

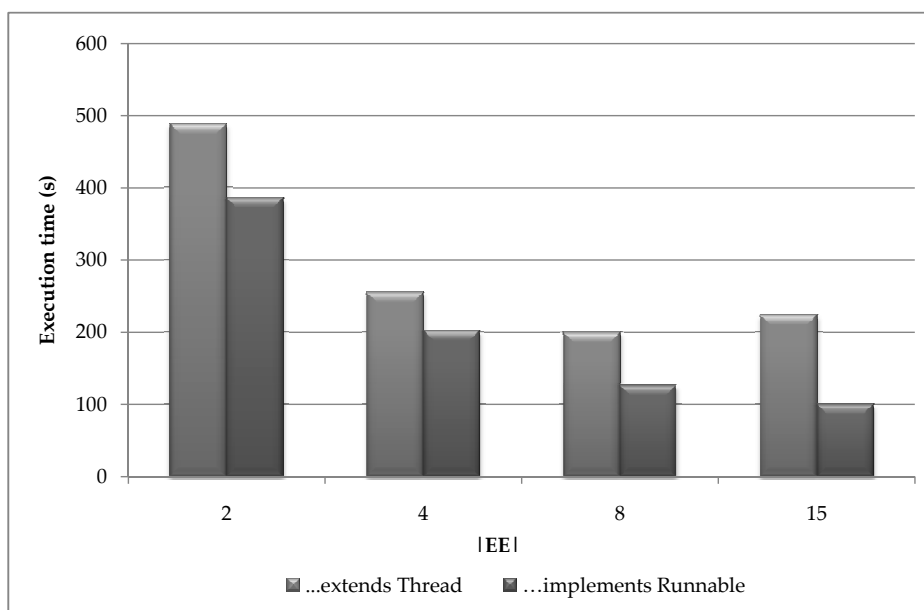


Figure 4.22. Comparison between extending Thread and implementing Runnable with $|E| = |A| = 512$, $\alpha = 4$ and $\beta = 1$

The reasons are twofold. On the one hand, the constructor of the Thread class is much more expensive than the very simple constructor of the Event class. This effect is even more noticeable since the benchmark test creates more than 150,000,000 events. On the other hand, a Thread object (or an object extending Thread) requires more memory than an object simply implementing Runnable. When such an object has to be moved to a different cache in a different processor, it entails a more severe penalty than a lighter object. Therefore, as the number of threads increments, so does the penalty of moving heavier objects.

4.12.3. Barriers

Barrier synchronisation is a common technique in shared memory parallel programming (Ball and Bull, 2003). Insofar as the execution of a parallel program can be divided into *episodes*, a *barrier* represents a synchronisation point where a set of threads must wait for each other before reaching the same episode.

After executing the events corresponding to simulation time ts , EEs have to wait for each other before advancing the simulation clock and executing the next events. Hence, the barrier implementations presented next offer a different alternative to the synchronisation mechanisms implemented in Subsection 4.12.1.

Java includes a centralised barrier (`CyclicBarrier`) in the `java.util.concurrent` package. However, the set of experiments conducted by the Communication Systems Group of the University of Bonn shows that this barrier has poor performance when compared to other designs, such as the Tournament barrier they incorporate in their simulator (Peschlow et al., 2009).

A Java library called `jbarrier` has been developed that adapts and comprises generic implementations of several barriers, such as tournament, butterfly and dissemination. The `jbarrier` algorithms support an *action* that can be executed *after* all the threads have reached the barrier and *before* the next episode has started. It is also possible to introduce *global reduction* operations. In order to illustrate the advantages of the barriers in `jbarrier` when compared to (`CyclicBarrier`), a small test is defined. The test consists of N threads that perform a certain amount W of workload and then synchronise at the barrier. This operation is repeated K times. Figure 4.23 shows the results for $W = 1024$ and $K = 1000000$.

All the barriers from `jbarrier` present similar execution times and good scalability as the number of threads increases. Contrarily, the inefficiency and poor scalability of `CyclicBarrier` is highlighted with this simple test. The overhead may be caused by the centralised scheme of the `CyclicBarrier`, which increments the contention of threads, and also by the amount of features offered by this barrier so to adapt it to any particular usage.

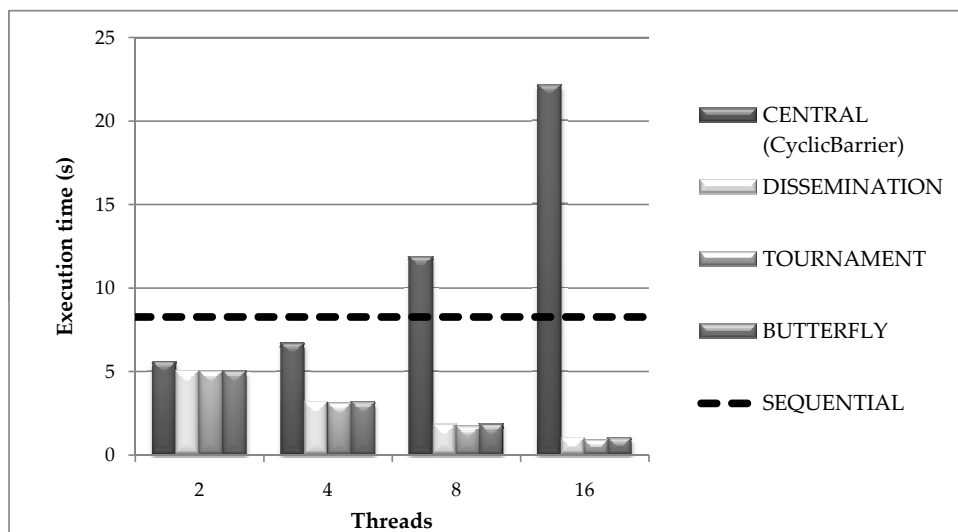


Figure 4.23. Performance of different implementations of barriers

As shown in Figure 4.23, all the implementations exhibit almost equivalent performance. PSIGHOS makes use of a tournament barrier, since the results obtained from many other tests corroborate that it usually shows the best execution time, especially as the number of threads increases. The associated *action* is a suitable place to execute the tasks originally assigned to the EM, that is, updating the VC, distributing events, updating the FEL, and so on.

Although the results obtained in Section 4.10 discourage retaking that path, the availability of a decentralised barrier with an associated action invites the use of a fully distributed simulation engine and forgetting about the EM. Figure 4.24 completes Figure 4.16 with this new implementation, and clearly illustrates that the scheme from Section 4.10 augmented with a well-configured barrier outperforms the previous approaches.

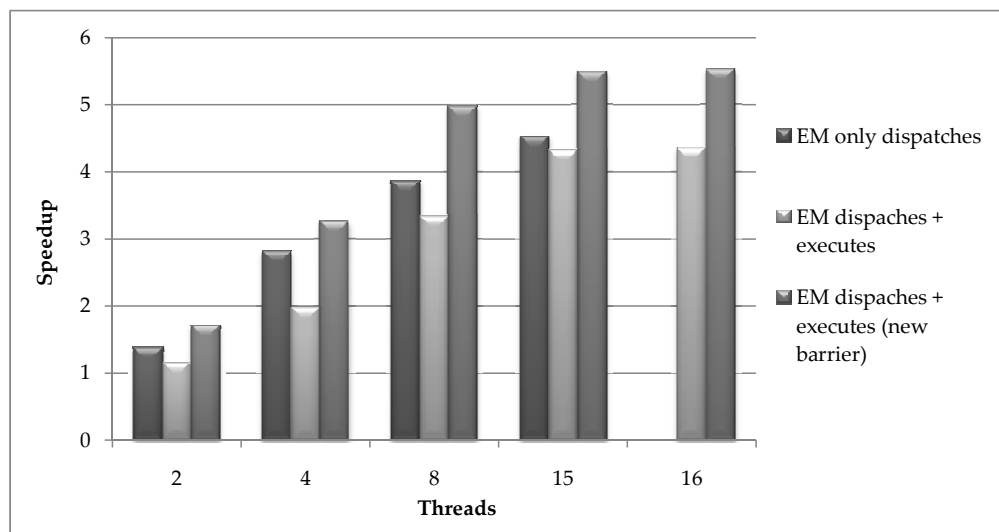


Figure 4.24. Speedup comparison among a pure master EM, a hybrid EME, and a hybrid EME using a tournament barrier for synchronisation

4.13. Putting It All Together: Hybrid EME like-3-Phase Approach

In the previous section it was shown that a well designed barrier beats other synchronisation strategies and boosts the hybrid EME architecture. Nevertheless, the best strategy in absolute numbers is the like-3-phase approach introduced in Section 4.11. The next logical step is to combine both ideas, as shown in Listing 4.19.

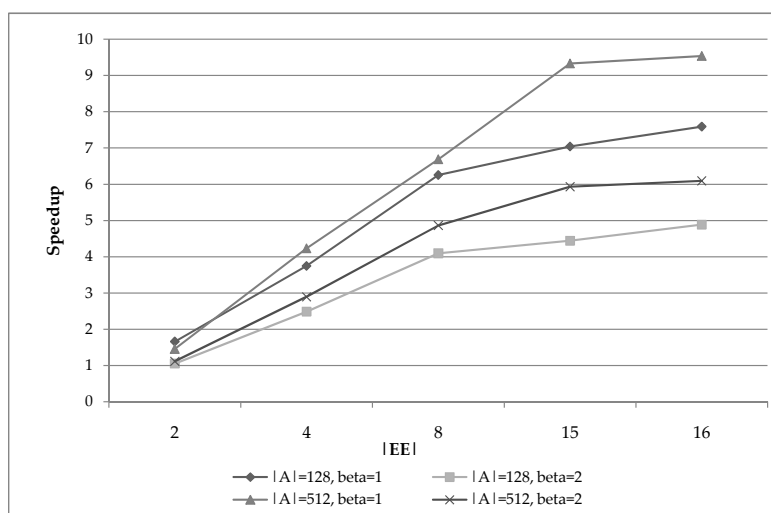
The speedup obtained with the modified algorithm can be seen in Figure 4.25. As expected, the results are very similar to those of the pure MS approach, but the possibility of using an extra thread offers an additional gain. Figure 4.26 directly compares both approaches and highlights the benefits of the additional thread.

Listing 4.19. Modified main loop of a hybrid EME like-3-phase Event Executor

```

while (Simulation.CST < Simulation.EndTs) {
  // Phase 2: Execute events
  E =  $\forall e@ts \in \text{Simulation.FEL} / (ts = \text{Simulation.CST})$ ;
  block = E.length / executor.length;
  executionCache = E[(id * block) .. ((id + 1) * block)];
  forall e@ts  $\in$  executionCache do {
    execute(e@ts);
  }
  // Synchronisation of Phase 2
  Wait until all the events have been effectively executed
  // Phase 3: AM Conditional execution
  forall ami  $\in$  (set of assigned AMs) do {
    execute(ami.Event@ts);
  }
  // Synchronisation of Phase 3: The action of the barrier will
  // update the VC and the FEL
  barrier.await();
}

```

**Figure 4.25.** Speedup for the hybrid EME like-3-phase approach with $|E| = 512$ and $\alpha = 8$

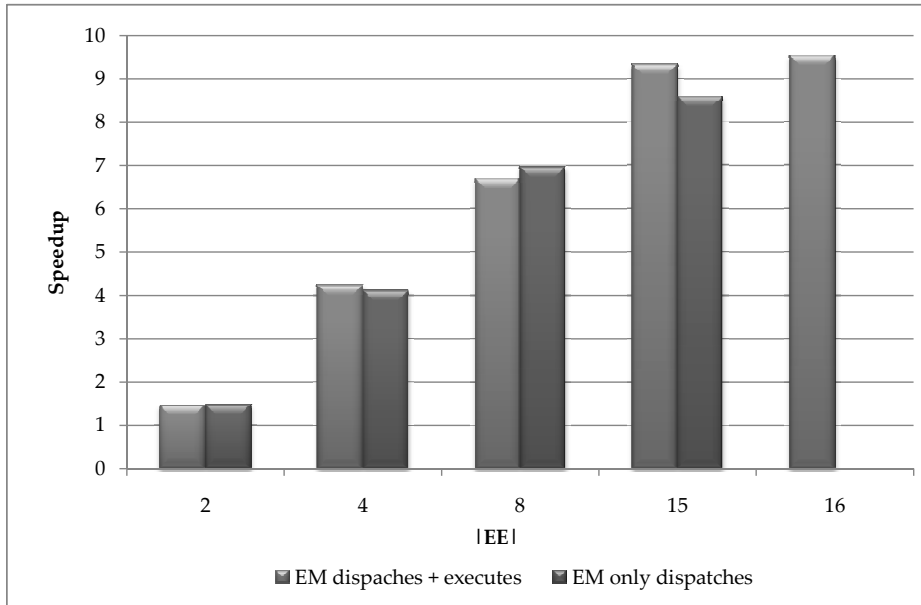


Figure 4.26. Comparison between the like-3-phase approach and its hybrid EME version with $|E| = |A| = 512$, $\alpha = 8$ and $\beta = 1$

In this case, the superlinear speedup already observed in Section 4.11 is effectively present, since $|EE| = 4$ means that exactly four threads are being used.

4.14. Summary: Comparing the Different Approaches

The previous sections of this chapter depicted the process used to obtain a parallel discrete event simulator that achieves the objectives enumerated in Table 4.1. Objectives G1-G3, related to transparency for the final user, were preserved by developing a solution based on a centralised list with distributed events, which changes the low level of the simulator without affecting the modelling strategies. Furthermore, an automated mechanism to identify and exploit the parallelism of the model, based on the creation of *Activity Managers*, was introduced.

Objective G4, related to efficiency, was successively refined by using several techniques:

1. Execution in a Java thread pool
2. Integration of the thread pool
3. Exploitation of event locality
4. Block dispatching
5. Redefinition of events and partition of the execution phase (like-3-phase algorithm)
6. Hybrid scheme EME
7. Fine-grain adjustment of implementation details

A final set of experiments was run to carry out a comprehensive comparison of the performance of PSIGHOS by applying the different optimisations, as shown in Table 4.8.

Table 4.8. Combination of parallel optimisations

Name	Techniques applied	Explained in...
EXT	1	Section 4.6
INT	2	Section 4.7
LOC	2+3	Section 4.8
BLO	2+3+4	Section 4.9
3PH	2+3+4+5	Section 4.11
HBL	2+3+4+6	Section 4.10
XBL	2+3+4+6+7	Subsection 4.12.3
H3P	2+3+4+5+6+7	Section 4.13

Table 4.9 shows the different combinations of parameters used to create the experiment scenarios. Each scenario was run either 10 iterations (when $\alpha = 4$) or 5 iterations ($\alpha = 8$).

Table 4.9. Parallel parameters

Parameter	Values
N	10,000
$ E $	512
$ A $	128, 512
α	4, 8
β	1, 2

Tables 4.10 and 4.11 summarise the average execution time obtained for each scenario.

Table 4.10. Final comparison of performance for $\alpha = 4$

$ A $	β	$ EE $	Execution time (s)							
			EXT	INT	LOC	BLO	3PH	HBL	XBL	H3P
128	1	2	81.276	65.496	70.274	59.916	30.117	71.504	43.680	28.867
128	1	4	97.458	54.173	42.131	33.081	16.775	40.562	24.280	16.676
128	1	8	164.230	73.461	27.898	20.092	12.198	29.167	20.773	11.666
128	1	15	197.499	77.026	23.235	17.621	11.241	24.998	18.726	11.332
128	1	16	-	-	-	-	-	25.196	18.524	10.887
128	2	2	95.921	89.181	93.579	93.100	45.699	182.071	71.696	46.129
128	2	4	90.972	58.202	56.489	45.975	24.440	123.247	36.694	24.361
128	2	8	150.702	71.798	35.380	28.587	16.857	68.557	24.771	16.580
128	2	15	207.492	74.143	27.739	20.390	15.491	33.821	22.882	15.247
128	2	16	-	-	-	-	-	41.514	22.102	14.753
512	1	2	83.193	65.257	59.588	43.366	41.367	67.778	41.149	41.177
512	1	4	98.771	55.034	34.251	24.396	18.897	41.319	23.876	18.583
512	1	8	167.439	72.176	24.100	17.888	12.984	24.463	17.888	12.783
512	1	15	195.413	75.897	19.938	16.274	11.703	19.432	17.064	11.645
512	1	16	-	-	-	-	-	19.313	18.568	11.654
512	2	2	90.729	83.827	79.747	63.619	58.894	99.359	64.678	53.410
512	2	4	102.540	51.627	44.061	29.886	25.803	62.352	33.993	25.360

Table 4.10 – Continued

A	β	EE	Execution time (s)							
			EXT	INT	LOC	BLO	3PH	HBL	XBL	H3P
512	2	8	148.936	71.381	27.782	20.084	17.447	30.531	20.019	17.015
512	2	15	201.161	74.423	22.164	17.285	15.842	23.243	17.889	15.455
512	2	16	-	-	-	-	-	23.424	18.255	15.116

Table 4.11. Final comparison of performance for $\alpha = 8$

A	β	EE	Execution time (s)							
			EXT	INT	LOC	BLO	3PH	HBL	XBL	H3P
128	1	2	126.507	130.347	134.960	134.350	52.676	125.936	78.609	58.126
128	1	4	85.043	77.204	76.672	76.399	24.307	73.972	41.809	25.179
128	1	8	133.186	69.689	48.307	35.453	16.779	49.993	38.005	16.810
128	1	15	202.476	76.528	35.848	28.293	16.070	40.251	29.146	15.974
128	1	16	-	-	-	-	-	43.684	30.494	15.460
128	2	2	181.685	192.436	187.117	216.119	94.691	318.678	127.931	91.102
128	2	4	110.176	106.576	109.916	110.933	39.522	215.936	65.921	40.609
128	2	8	128.476	69.005	66.614	57.448	25.550	121.440	38.997	25.819
128	2	15	215.699	80.691	49.515	36.442	23.397	62.092	36.983	23.603
128	2	16	-	-	-	-	-	77.281	39.238	22.139
512	1	2	112.570	119.151	110.181	85.018	81.933	124.054	77.001	82.394
512	1	4	97.663	63.314	59.101	38.875	27.188	66.635	39.507	27.100
512	1	8	134.958	72.254	39.274	28.991	18.558	39.823	28.981	18.295
512	1	15	203.964	73.399	32.546	26.776	16.547	32.595	28.203	16.198
512	1	16	-	-	-	-	-	32.406	27.939	16.242
512	2	2	126.641	148.034	139.255	112.697	105.865	175.084	132.816	116.982
512	2	4	88.203	75.947	73.102	52.978	42.423	104.214	65.079	42.084
512	2	8	132.770	66.194	50.261	34.577	26.951	59.122	41.282	27.256
512	2	15	208.307	76.038	35.929	28.173	23.462	41.056	29.840	23.555
512	2	16	-	-	-	-	-	41.325	29.620	22.785

If we consider the first approaches, the longest gap in absolute values appears between the external pool (EXT) and the internal one (INT). However, the highest relative reduction in execution time is brought about by exploiting of event locality. Furthermore, the simulator starts being scalable only after applying this technique.

The like-3-Phase algorithm exhibits the best behaviour for all the scenarios. More specifically, the hybrid EME version of the algorithm (H3P), by being able to take full advantage of the available processors, achieves the best performance. Among the approaches using the original algorithm, only BLO and XBL achieve a reasonably good performance. Nevertheless, the 8-processor (and sometimes the 4-processor) like-3-Phase version outperforms the best results obtained with these approaches when all the available processors are utilised.

Table 4.12 shows a more in-depth analysis of the speedup of the H3P version of the simulator.

Table 4.12. Speedup for H3P

Threads	α	4				8			
	$ \mathbf{A} $	128		512		128		512	
	β	1	2	1	2	1	2	1	2
2		1.601	1.168	1.952	1.608	1.574	1.233	1.491	1.110
4		2.772	2.212	4.325	3.386	3.633	2.766	4.534	3.086
8		3.962	3.250	6.288	5.047	5.441	4.351	6.716	4.764
16		4.246	3.653	6.897	5.680	5.916	5.074	7.565	5.699

The results above confirm that PSIGHOS can handle several scenarios generically. However, the characteristics of the different scenarios being modelled have a significant impact on the final performance:

- PSIGHOS requires a considerable rate of simultaneous events. Hence, the more complex and expansive the model is, the more noticeable the performance gain will be.
- Models incorporating complex user actions within their events also benefit. Bare events are extremely lightweight tasks, so adding extra complexity will improve the computation/communication rate of the tool. This is observed by comparing the speedup for $\alpha = 4$ and $\alpha = 8$, especially when more threads are involved.

- Loosely coupled systems, where resources have specific uses, and entities carrying out specialised activities will benefit from PSIGHOS approach. Obviously, the best results are obtained for scenarios with the maximum feasible amount of parallelism ($|A| = |E|$ and $\beta = 1$).

Chapter 5

Case Study: A Model for Hospital Management

The synthetic test benchmark described in Section 4.4 is suitable both to verify the correct behaviour, and to measure the performance of PSIGHOS. Nevertheless, a more realistic model is required to prove the usefulness of the tool. This chapter will present a prototype of a generic full hospital model. The model is not intended to be an accurate representation of a real hospital, but a sufficiently realistic proving ground for both 1) the expressive power of PSIGHOS and 2) the potential advantages of the parallel approach when modelling a real system.

5.1. Overview of a Hospital

Hospitals, whether publicly or privately financed, are service-oriented organisations devoted to the diagnosis and treatment of patients.

Hospitals can be classified according to several parameters, such as size, scope or financing. Another classification used, as in Spain, for example, focuses on the health care target. Hence, there are hospitals that cater mainly to surgical procedures, paediatrics or psychiatry. *General Hospitals* attend to patients with a variety of pathologies, and comprise medical and surgical diagnosis and treatment, obstetrics, gynaecology and paediatrics. The term *General* is also applied to hospitals where one or more of these areas have been scarcely developed,

whenever none of the remaining areas concentrate the resources and efforts of the institution.

The head of a hospital is the *Manager*, who is assisted by several advisors, and typically assigns sub-managers to different areas of the hospital, such as human resources, nursing, medical, etc. For example, Figure 5.1 shows the organisational chart of the Hospital Universitario de Canarias (HUC).

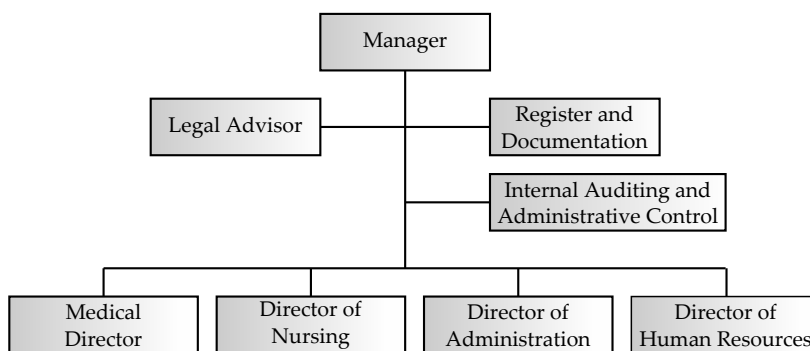


Figure 5.1. HUC's organisational chart

Hospitals are generally organised into *departments* according to specialties, such as rheumatology, dermatology, neurosurgery, urology, traumatology, and others. Departments, in turn, are divided into *units*.

Besides the specialised departments, hospitals also provide diagnostic tests (such as X-rays, scans, ultrasound, and nuclear medicine) and laboratory services (such as haematology, microbiology and anatomopathology). These services are generically referred to as *central services*.

Patient appointments and surgical operations take place in surgeries and operating theatres (OTs) respectively. In general, surgeries are associated with specific departments, whereas OTs can be shared by several departments.

Each department allocates its admitted patients into wards. Other specialised units including beds are Intensive Care Unit (ICU), Postanaesthesia Care Unit (PACU), the elderly care services, and others.

Patients are directed to one department or another depending on their pathology. However, patients are not categorised based solely on the pathology. For example, a clear distinction is made between *emergency* and *elective* patients. The former are attended to in the Accident and Emergency (A&E) department, whereas the latter voluntarily receive diagnosis and treatment from another medical or surgical department. Furthermore, patients who are admitted and stay in a hospital bed to receive medical or surgical treatment are called *inpatients (IPs)*; whereas patients having an appointment with a doctor, getting a diagnostic test or receiving a treatment not requiring an over-night stay are termed *outpatients (OPs)*.

Although each department comprises its own material and specialised human resources, most departments interact with each other and with other services. For example, every department depends on central services, and A&E not only requests diagnostic tests from central services, it also sends patients to be admitted by other departments.

5.2. Hospital Performance: Why Modelling and Simulation?

Hospitals and, in general, healthcare institutions require substantial funding. The management of such organisations involves smart and careful decision making processes, especially in healthcare systems such as Spain's, which are financed through taxation and are free to all citizens. The financial support received is intended to improve the quality of service at the hospital, which results in management's need to properly measure the performance of the organisation they are overseeing.

According to Neely et al. (1995), a performance measure is

« ... a metric used to quantify the efficiency and effectiveness of an action. »

Contrarily to other systems, such as manufacturing, where finding suitable Performance Indicators (PIs) to quantify the performance of the organisation is

straightforward, hospitals present additional difficulties to measuring effectiveness or efficiency. The complexity and size of the organisation have a strong influence, but the lack of a properly identified “production” parameter aggravates the problem.

Among the possible PIs, *wait times* are generally considered a reasonable choice for managers. Both the Quality Plan for the National Health System of Spain (SNS, 2007) and the UK National Health System (NHS) take into account this parameter. The perception that patients have of the quality of the service depends on many other factors, but these are probably more subjective and difficult to quantify.

Due to the interaction between the different hospital departments and services, trying to measure wait times by considering a single department or service will probably prove erroneous. Hence, the significance of highlighting the usefulness of a holistic view of a hospital, where the most important relationships among units are carefully considered.

Hospital management requires comprehensive tools that aid in understanding the complex relationships present in their organisation and provide valuable information to the decision-making processes. This provides the perfect setting, then, for a modelling and simulation scenario.

5.3. Background

Some of the reasons supporting the selection of a hospital as a case study have already been exposed. However, another relevant factor in making this choice is the strong background that the Simulation Group from the Universidad de La Laguna has in this area. This group has been working on simulation as a tool to aid in hospital management for many years. Indeed, *SIGHOS*, as already stated in Section 3.3, was originally planned as a hospital simulation tool. Only later was it extended to generically handle all kinds of organisations.

Within this context, the line of research followed by our Simulation Group is motivated by two main factors:

1. The strong need to accelerate and justify the effectiveness and efficiency of the actions performed by hospital management.
2. Cost reduction through optimising the use of existing resources.

Although there were some existing preliminary contributions, this line of research was formalised in (Aguilar, 1998). Aguilar focuses on the development of a decision aiding tool that comprises three subsystems (see Figure 5.2):

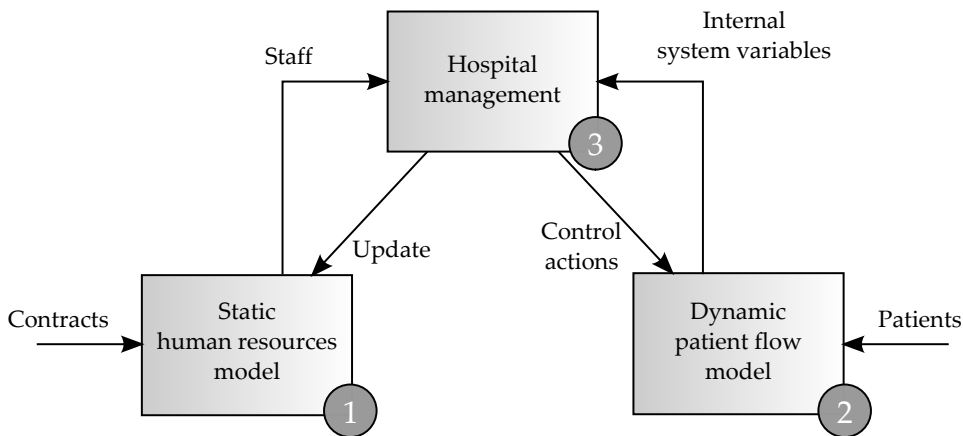


Figure 5.2. Basic diagram of the decision aiding tool. Source: (Moreno et al., 2000)

- *Subsystem 1* is a database with the hospital resources.
- *Subsystem 2* is a tool that simulates the patient flow through the various hospital departments, and the interaction of those patients with the available resources. Developed using Modsim (CAC, 1995), this subsystem is a process-oriented DES precursor of SIGHOS.
- *Subsystem 3* is a knowledge-based system consisting of the manager's heuristic knowhow on how to distribute and handle resources.

This tool is intended to work with the hospital's information systems in order to obtain meaningful information about the hospital's current status. Hence, the manager obtains further information about the behaviour of the organisation. Moreover, the tool can predict the effect of adding new departments to the hospital, as well as their impact on existing ones (especially on central services).

This line of research, aided by our close collaboration with the management of the Hospital Universitario Nuestra Señora de Candelaria (HUNSC), has led our research group to publish papers in several journals (Aguilar et al., 2005b, 2009; Moreno et al., 2000, 2003), as well as to contribute to international conferences (Aguilar et al., 2005a, 2006; Castilla and Aguilar, 2009; Castilla et al., 2008).

5.3.1. First Steps: a Model of the General and Digestive Surgical Department of the HUNSC

As part of the Simulation Group, the author of this thesis has been working alongside the staff of the HUNSC to analyse information systems by attempting to find suitable sources and accurate data for the simulation models. Specifically, a national award-winning simulation model to 1) analyse and 2) predict the behaviour of their surgical departments was presented in (Bermúdez et al., 2008). Since the hospital model that will be presented later in this chapter is partially based on this collaboration, further comments on this topic are warranted.

Some time ago, HUNSC management started scheduling surgeries in the afternoon shift in an effort to reduce waiting lists. The outcome of this measure was not as positive as expected. Hence, a simulation model was needed that could answer the following questions:

1. How do flexible OT timetables affect patient wait times?
2. Can the system deal with surgeries scheduled in the afternoon? How much would it cost? If the answer to the first question is affirmative, why was the real system not able to do it?

The simulation model focuses on a single department, the *General and Digestive Surgery (GS) department*, which exhibits the typical problems of that hospital's departments. Figure 5.3 shows the basic workflow of patients who have to undergo surgery.

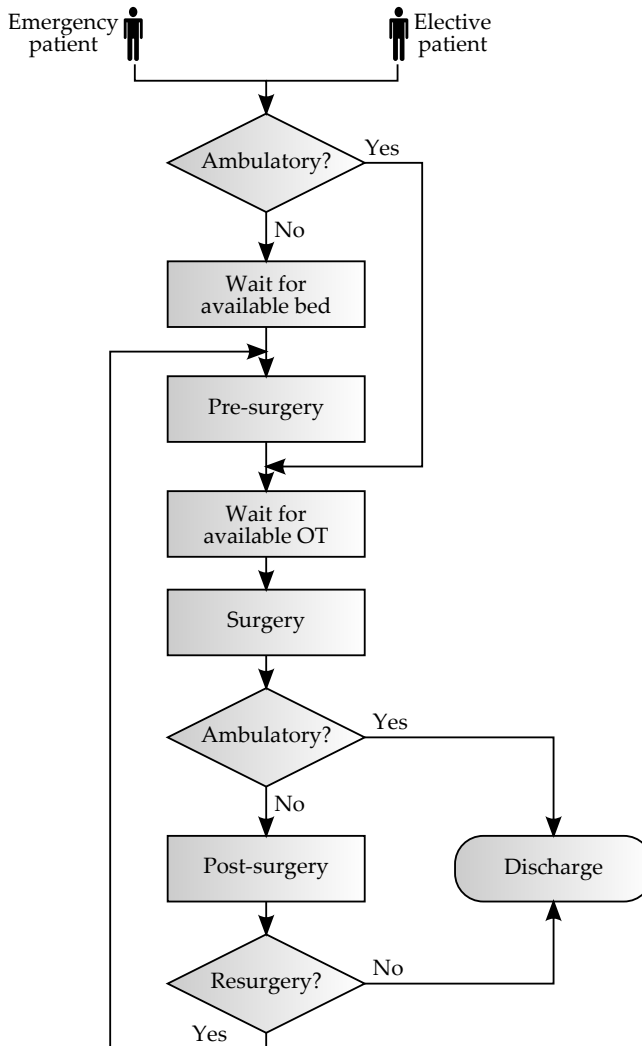


Figure 5.3. Basic schema for surgical patients at the HUNSC

This schema assumes independent surgical procedures (thus neglecting the effect of the patient's current pathology on his/her future state), and avoids, for the sake of simplicity:

- The admission procedure. All patients are assumed to require surgery.
- Any previous or subsequent stay in observation or recovery units.
- Resurgery for ambulatory patients.
- *Exitus* (death), transfers, etc. The *discharge* stage will involve all these cases.

The conceptual model involves only two resource types: OTs and beds. A different and perhaps more detailed approach would take into account the surgical team but, by carefully managing the OT timetable, an upper bound for the hospital's real capacity can be obtained.

With respect to beds, the model only needs the number that are assigned to GS. OTs are treated in slightly more detail, and can be categorised into:

- *Emergency (E)*, devoted to emergency surgical procedures.
- *Ambulatory (A)*, devoted to ambulatory elective surgical procedures.
- *Scheduled (S)*, devoted to non-ambulatory elective surgical procedures.

These categories are not exclusive, that is, scheduled OTs can host emergency surgical procedures if required, and vice versa. Even ambulatory OTs can be used for non-ambulatory surgery from time to time.

The computational model for the workflow described above is implemented in SIGHOS and is fed with data extracted from the HUNSC information systems. The simulation experiment can be defined as an input/output schema, as shown in Figure 5.4.

According to their databases, HUNSC uses six OTs for GS. The preliminary schedule shown in Table 5.1 is used to determine the weekly usage of each OT.

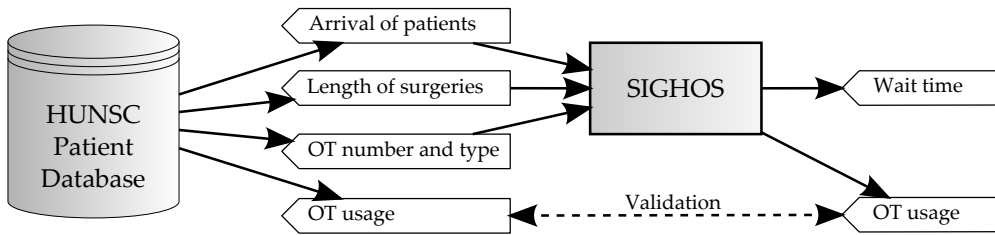


Figure 5.4. IO schema of the simulation model

Table 5.1. Preliminary schedule of operating theatres for GS

Day of week	Operating Theatre					
	3-1	3-5	3-6	4-5	4-6	AMB
Monday	-	E	E	S	S	A
Tuesday	-	E	E	S	S	-
Wednesday	-	E	E	S	S	-
Thursday	-	E	E	S	S	-
Friday	S	E	E	S	S	-

This preliminary schedule can be contrasted with the real usage of the OTs for the 2004-2005 time period. A careful study of the length and scheduling of the surgical procedures at the hospital offers many insights into the proper modelling of such a system. For example, Table 5.2 shows the total number and length of elective surgical procedures in the morning shift. The *correct usage* row represents the percentage of elective surgical procedures that match the preliminary schedule. The percentage of *wrong usage* corresponds to scheduled surgical procedures taking place in the wrong time slot, as defined by being either not previously scheduled or devoted to a different purpose. A surgical procedure is considered *exogenous* if it is performed in an OT different from the planned OT.

As we can see, more than 90% of the usage matches the preliminary schedule. The main divergence occurs with ambulatory surgical procedures, since

Table 5.2. Real usage of OTs: Elective surgical procedures in the morning shift 2004-2005

	Ordinary		Ambulatory	
	Number	Length (m)	Number	Length (m)
Total	2,837	653,318	824	34,718
Correct usage	95.62	97.65	94.20	90.59
Wrong usage	2.93	0.67	5.68	8.95
Exogenous	1.45	1.68	0.12	0.46

more than 5% were performed in the wrong OTs. Being simpler procedures, the person in charge of arranging the final schedule probably filled the gaps in the non-ambulatory (ordinary) schedule with ambulatory surgeries.

The same validation process can be applied to emergency surgical procedures and afternoon-shift elective surgical procedures. The idea is to check the suitability of using the preliminary schedule in the computational model, thus establishing a direct link between the patient type (elective - emergency, ambulatory - non-ambulatory) and the OT used to carry out the patient's surgery.

Both the arrival of patients and the length of the surgical procedure can be obtained by carrying out a statistical analysis of the hospital's databases. For the sake of simplicity, interarrival times are assumed to be independent and exponentially distributed. With respect to the length of the surgical procedures, a curious effect appears: since data are collected by humans, they normally round the samples to the nearest five-minute multiple. Different lengths can be associated with each patient type.

A Microsoft Excel interface complements the simulation model developed. The interface allows a user with a limited knowledge of simulations to parameterise the model and create different "what-if" scenarios. For example, a scenario with the OTs defined in Table 5.1 could be compared to a new scenario where one of the OTs can be used to perform both ordinary and ambulatory surgical procedures. Figure 5.5(b) and Figure 5.5(a) show the wait time for ordinary and ambulatory surgeries, respectively. Note how ordinary surgeries are not affec-

ted by the change, but a considerable reduction in the wait time is obtained for ambulatory patients.

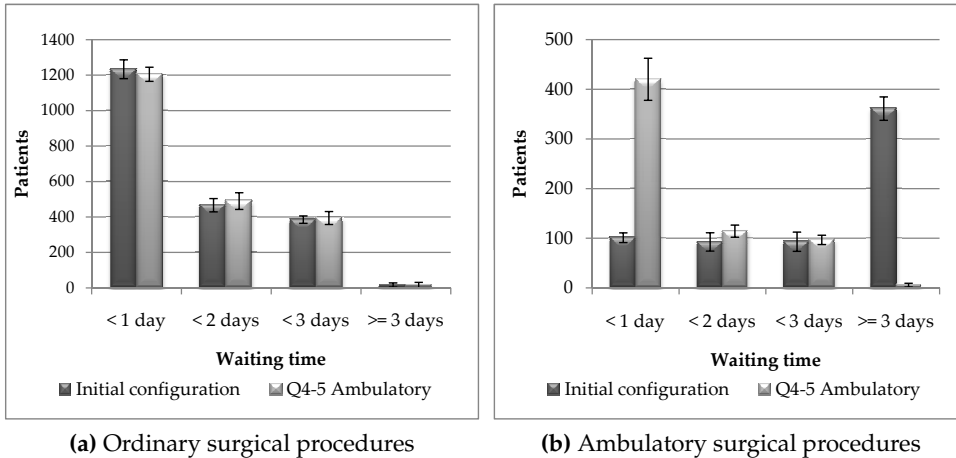


Figure 5.5. Comparing two scenarios with different OT configurations

5.4. Review of the Literature on Modelling a Whole Hospital

Although we are not interested in the formal validation of the test model, it would be highly desirable to obtain a prototype that could later be improved and further developed so to achieve a more valuable tool.

In this sense, a review of the literature on modelling of a whole hospital is required. Fortunately, a PhD Thesis from the University of Lancaster (Günel, 2008) includes a comprehensive, if compact, review of the literature on this topic. Such a review will be used as the basis for this one.

There are several comprehensive surveys on modelling and simulation in the area of health care. To cite some examples, Fone et al. (2003), Jun et al. (1999) and very recently Mustafee et al. (2010) thoroughly and systematically cover the related literature. Another noticeable reference, though not formally published, is the Master's Thesis of Alla (2005), who analyses a considerable num-

ber of papers and simulation projects spanning from 1990 to 2005. The different simulation studies are classified according to the scope of the model, the type of questions posed, level of detail, data sources, software, Verification and Validation (V&V) techniques, and other aspects.

Focusing on the scope of the models, all these reviews agree that the A&E department is the most common objective for simulation studies, followed by individual departments, clinics and, finally, whole hospitals. Günal and Pidd (2005) state that the lack of simulation studies comprising the whole hospital is evidence of the complexity of such a system: hospitals consist of many interactive components, which makes measuring the final performance a difficult and subtle exercise in combining outcomes from different sources.

Aguilar's hospital model from is one of the few simulation studies that actually deals with the modelling of a whole hospital. A different approach is described in (Brailsford et al., 2004), who do not use DES, but rather a System Dynamics (SD) model of emergency and on-demand health care in Nottingham, UK, which represents the flow of patients through different departments in the hospital. Although, in light of the requirement for a holistic and dynamic system model, SD seems like a natural alternative, Günal and Pidd (2008) defend the use of DES due to the importance of stochastic effects in congested systems.

Cochran and Bharti (2006) present a two-stage methodology for balancing bed unit utilisation in a 400-bed hospital. The first stage uses Queuing Network Analysis (QNA) to find an optimal bed reallocation distribution at peak demand. Stakeholder targets, priorities and constraints are taken into account during this first phase. The aspects that QNA cannot handle, such as time dependent arrival patterns and non-exponential lengths, are evaluated during the second stage using DES.

Meer et al. (2005) focus on reducing the wait times for elective patients, both for a first outpatient appointment and for the subsequent inpatient treatment. Their model is specific to orthopaedics, but is very detailed, and covers every stage involving an elective patient.

Finally, Günal and Pidd (2008) present DGHPSim, a suite of DES models for evaluating wait time performance in hospitals in the UK NHS. DGHPSim comprises 4 interconnected submodels (see Figure 5.6):

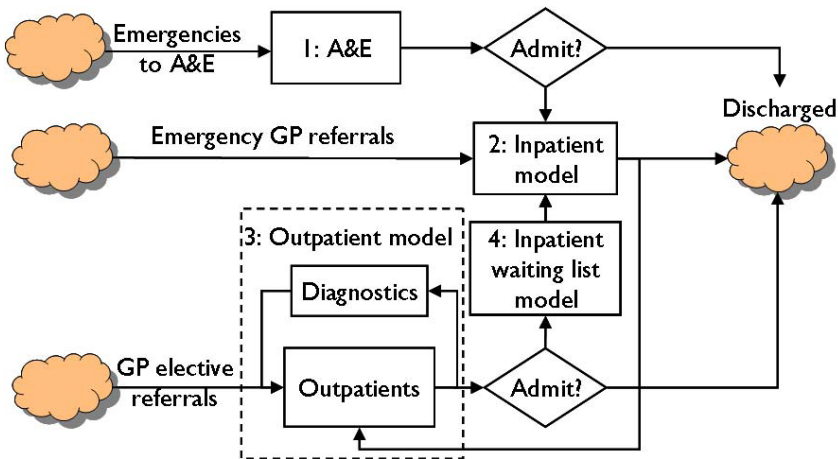


Figure 5.6. Hospital Model. Source: (Güenal and Pidd, 2008)

- *A&E department* is the main entry point into a hospital for emergency patients. Both walk-in and ambulance arrivals are taken into account. Registration, triage and tests are considered as part of the patient flow before admission or discharge. Güenal and Pidd (2006) present further results by using this model.
- The *inpatient* submodel focuses on the utilisation of the main wards of a hospital, including patients moving between different wards. Patients proceeding from both the A&E department and the waiting lists are considered.
- The *outpatient* submodel considers patients after they are referred from General Practitioners (GPs). Patients then have one or more outpatient appointments, diagnostic tests, and, in some cases, are placed on the waiting list.

- The *waiting list* submodel is used as a bridge between inpatients and outpatients. In essence, this submodel consists of a set of rules that define the way inpatients are admitted from the outpatient submodel.

All but the last submodel can be run independently, but a complete execution provides a holistic view of the hospital's performance and allows a manager to answer questions related to the expected wait time for a given investment.

DGHPsim is a generic model that is parameterised to adapt to a particular hospital by means of two data sources: local data from the hospital's Patient Administration System (PAS); and the Health Episode Statistics (HES), collected nationally by the UK Department of Health. Those data sources cannot be used directly, and require a specific software to transform raw data into meaningful and structured information.

5.4.1. HADA: Hospital Activity Data Analyser

A standalone software program called HADA (Castilla et al., 2008) has been developed by the author of this thesis together with Dr. Murat Günal to analyse both PAS and HES data for understanding a hospital's past performance, as well as for estimating parameters of a hospital simulation model (such as DGHPsim). Figure 5.7 schematises the interrelation between the application and the DGHPsim modules.

HES analysis

HADA embodies one software module per data source. The first works with HES, a UK-wide, routinely collected dataset capturing details of all inpatient and outpatient hospital episodes in the NHS. HES includes two huge datasets: one for outpatients (HESOP), which includes data on all outpatient appointments; and one for inpatients (HESIP), which considers decisions to admit, admission and operations. HESIP and HESOP structures are defined in the HES-Online Data Dictionary (The NHS Information Centre, 2008).

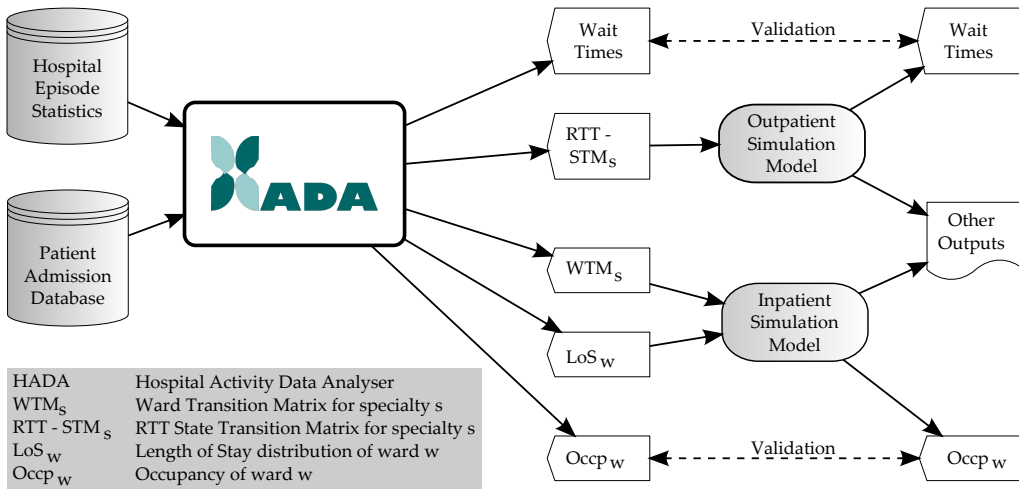


Figure 5.7. HADA. Source: (Castilla et al., 2008)

Though these data provide a general picture of the Referral-To-Treatment (RTT) path taken by a hospital patient, there is no unique identifier matching registers from HESIP and HESOP. HADA identifies some key fields and utilises heuristic knowledge to link events from both datasets in order to create a complete path. Based on the events of the path, HADA computes patient wait times and the RTT state transition matrix. The latter is input to the outpatient submodel, whereas the former is used for validation.

PAS analysis

Hospital information systems provide more detail on the specific characteristics of the organisation. Nevertheless, being specific to each particular hospital, PAS data are generally heterogeneous, and errors and inconsistencies appear more frequently than in HES.

HADA's PAS module relies on the user to identify some key fields in the hospital's local databases, such as patient identifier, spell number, patient classification, ward code and admission date. Then, HADA computes three different

outputs: bed occupancy, length of stay and transitions between wards. With respect to wards, they can be considered independently or grouped, thus allowing the user to control the model's level of detail by combining similar wards.

5.5. A Modular Model for a Whole Hospital

From the review of the literature, it is clear that developing a detailed and realistic hospital model is an extensive task that is beyond the scope of this thesis. We will limit ourselves to presenting a model that combines some of the findings from Aguilar (1998), Bermúdez et al. (2008) and Günal (2008).

Even when the model is not intended to be actually utilised in a real simulation study, some of the model's basic objectives must be established in advance. These objectives will help the modeller to focus on the relevant characteristics of the system and to orient the decisions made during the design of the conceptual model. Therefore, let the aim of the model be the study of the patient flow through the departments of the hospital, and how different resource configurations affect patient wait times.

The purpose of the simulation study is to exploit the modelling advantages of PSIGHOS. The strong focus on patient flows and resources not only indicates the suitability of a process-oriented approach, but it is intended to benefit from the expressive power of the tool. The *functional* goals of the model can be summarised as follows:

- To serve as a basis for more realistic models in the future
- To offer an easily customisable scale and level of detail
- To be complex enough to provide a high rate of simultaneous events

The PSIGHOS implementation of the hospital's computational model must be preceded by the development of a conceptual model. The conceptual model helps the modeller to better understand the system under study; to shape and set the objectives or *main questions* that the simulation study is intended to answer;

and to clearly identify the inputs, outputs, entities and attributes of the system, as well as its boundaries.

5.6. Conceptual Model

Following the steps of Günel (2008), a *divide and conquer* approach is utilised in order to deal with the complexity of a system like a hospital. In the schema already presented in Figure 5.6, Günel and Pidd apply the divide and conquer strategy to split a patient's entire path into different stages, each of which become a submodel. Hence, a patient can start in the A&E submodel, then pass to the inpatient submodel before finally being discharged. A different patient can be referred by a GP and start in the outpatient submodel, then pass to the waiting list submodel and eventually be admitted as an inpatient.

We propose a similar approach where different *functional* modules are defined. Since we want to stress the importance of resources, our division will take them into account when splitting the model. Therefore, we will define four modules.

- The *A&E department* receives emergency patients, both walk-in and ambulance arrivals. Some of the patients are subsequently discharged, but others are sent to the corresponding medical or surgical department.
- The *Surgical Departments (SDs)* module comprises all the SDs of the hospital, as well as some common units and facilities, such as the PACU and ICU. The only interaction between SDs involve the utilisation of these common facilities. Patients with pathologies that may require surgery are input to this module. Both emergency patients from the A&E department, and elective patients referred by a GP are attended to.
- The *Medical Departments (MDs)* module is defined analogously to the SD. Instead of patients with pathologies requiring surgery, they attend to patients requiring a medical diagnosis and treatment.

- *Central services* not only attend to requests from any other module, but from peripheral centres.

Figure 5.8 shows a high-level block diagram with the interrelations among the different basic modules. Modularity allows a modeller to focus on the specific needs of each submodel. Furthermore, each submodel can be considered as a black box, which allows a simulation user to replace said submodel with an equivalent probability distribution to serve as a patient generator, as noted by Günal and Pidd (2007). The result is a model where the level of detail can be reduced in favour of a faster execution time.

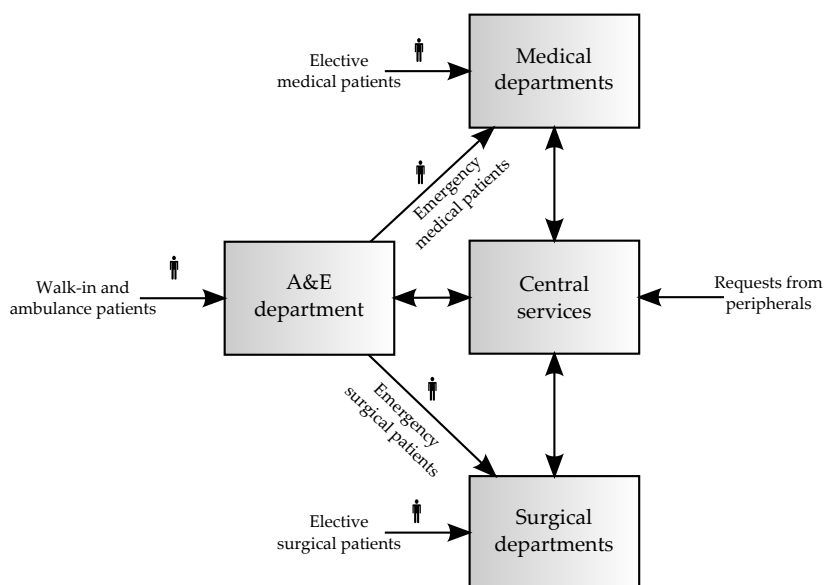


Figure 5.8. High-level conceptual model of a hospital

This model focuses on the patient flow through the various hospital departments. Hence, patients will be treated as individual entities. However, different paths can appear for the same patient due to the simultaneous presence of several pathologies. Our approach considers each path as a different patient be-

cause a detailed model of the interaction of the pathologies on the same patient is beyond the scope of this study.

The decision to always treat patients as individual entities within the model is arguable. For example, due to the limits imposed by Micro Saint Sharp, Günal's approach to the outpatient submodel substitutes the individual patients by a *patient-state changing machine*. Not only this, since his simulation study is not interested in the in-clinic waiting times, he argues that a higher level of detail is not required. Given that our main concern is to put the performance of the simulation tool to the test, we will not simplify the model in this sense.

The following sections will describe each module or submodel, except for the A&E department, which has been omitted from the final model. The absence of this module stems mainly from time restrictions in the development of the model. As noted earlier, the A&E department could be substituted with some patient generators. Nevertheless, the final decision was even more drastic, and only elective patients have been taken into account. This assumption clearly prevents the model from being fully validated, but leads to faster development. At the same time, removing the emergency patients limits the accuracy and realism of the model, but does not preclude the model from being used to check the performance of PSIGHOS.

The definition of this conceptual model is based on numerous visits and interviews with staff from the HUNSC, and has been validated with staff from the HUC.

5.6.1. Medical Departments

The typical patient flow in a MD can be schematised as shown in Figure 5.9. After referral, elective medical patients have an initial appointment with a doctor in a surgery. The doctor examines the patient and makes a decision, which may include: 1) discharging the patient; 2) ordering diagnostic tests on the patient for review in a follow-up appointment; 3) admitting the patient for medical treatment. The same decisions apply after each follow-up appointment. A

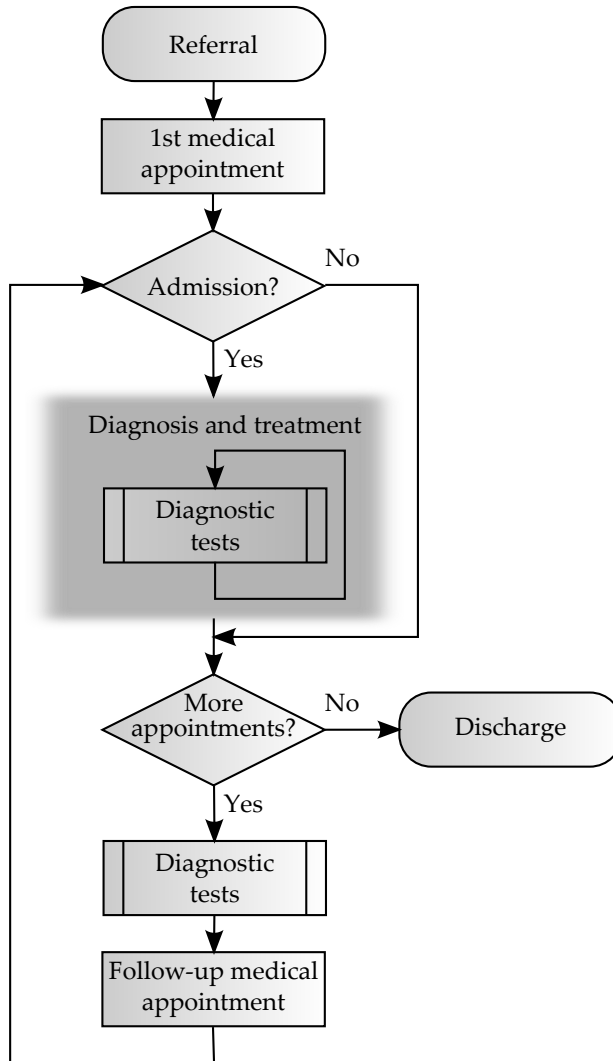


Figure 5.9. Flow for a medical patient

patient may also be redirected to a different department, or even to another hospital. However, in this case, a new path (and consequently, a new “patient”) would appear.

5.6.2. Surgical Departments

The patient flow in a SD, as shown in Figure 5.10, shares some similarities with the MD patient flow. The main difference involves the path of surgical inpatients. Here, a distinction is made between three different types of surgeries: *ambulatory*, for patients who are admitted and discharged from the hospital on the same day; *short-stay*, for patients who require a bed for a short period, that is, 1-2 days; and *ordinary*, for all other patients.

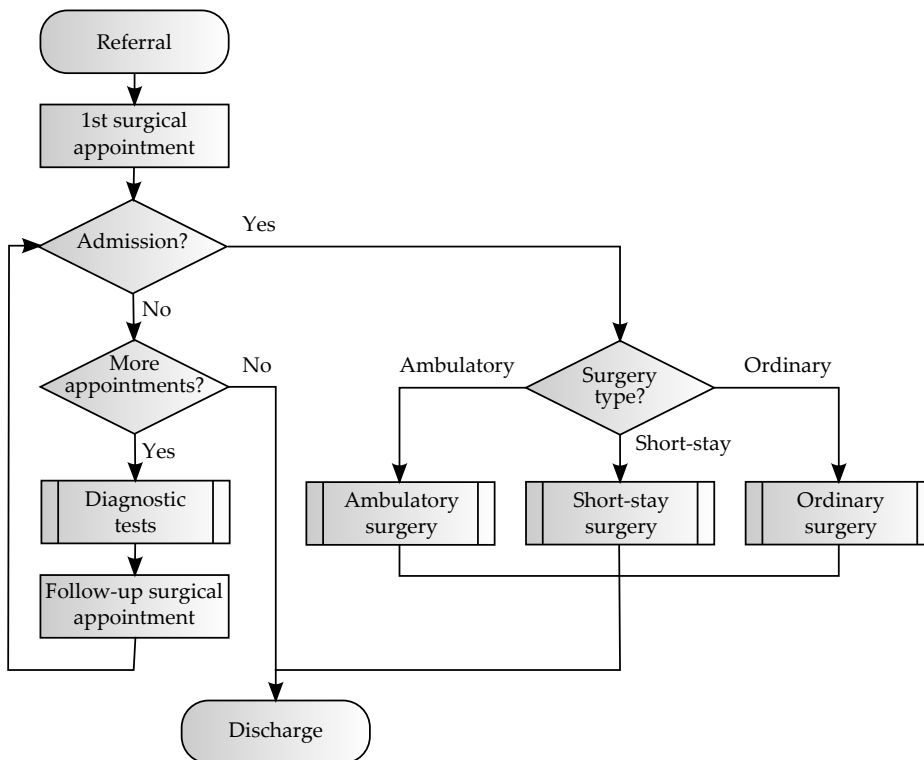


Figure 5.10. Flow for a surgical patient

A subtle difference between the conceptual model for medical patients appears after a patient undergoes surgery: whereas medical patients (and more

specifically, chronic medical patients) may continue their path after being admitted, surgical patients are always discharged. We assume that patients who start again with an outpatient appointment are new patients, that is, a patient's path is considered finalised as soon as the patient undergoes surgery.

Ambulatory patients follow the simplest path (Figure 5.11(a)). They undergo surgery and recover in the PACU before finally being scheduled for attend for a post-operative appointment. Requiring resurgery, though possible, is very rare, and has not been considered in the model.

Short-stay patients require a bed in the hospital (generally, a limited number of beds is reserved for these patients). Usually, a few diagnostic tests are ordered before the patient leaves the hospital. Once again, resurgery is ignored. Figure 5.11(b) schematises the main steps of the trip for a short-stay patient.

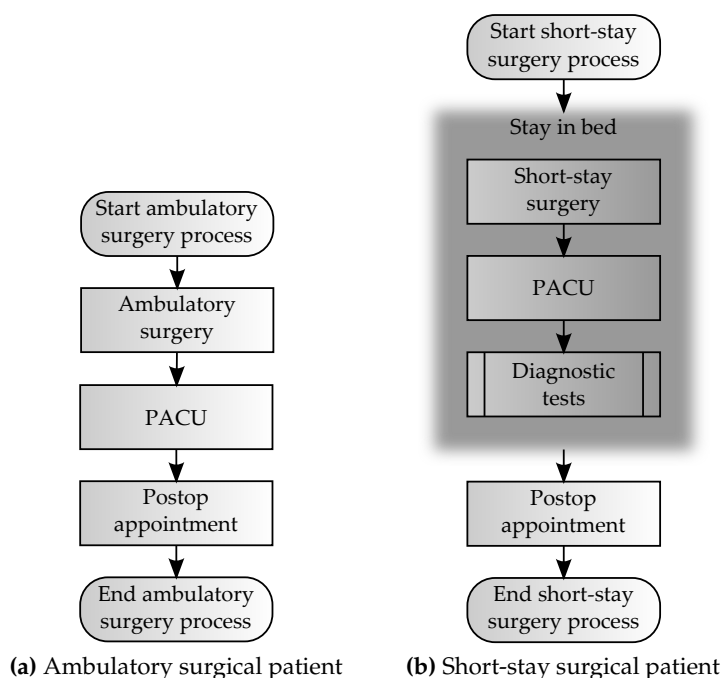


Figure 5.11. Flow for ambulatory and short-stay surgical patients

Ordinary patients also require a bed but, being their stay longer, their treatment involves more frequent tests. Furthermore, since we are talking about more complex surgical procedures, the probabilities of being passed to the ICU or requiring a resurgery increase. Figure 5.12 depicts the flow for a patient in need of ordinary surgery.

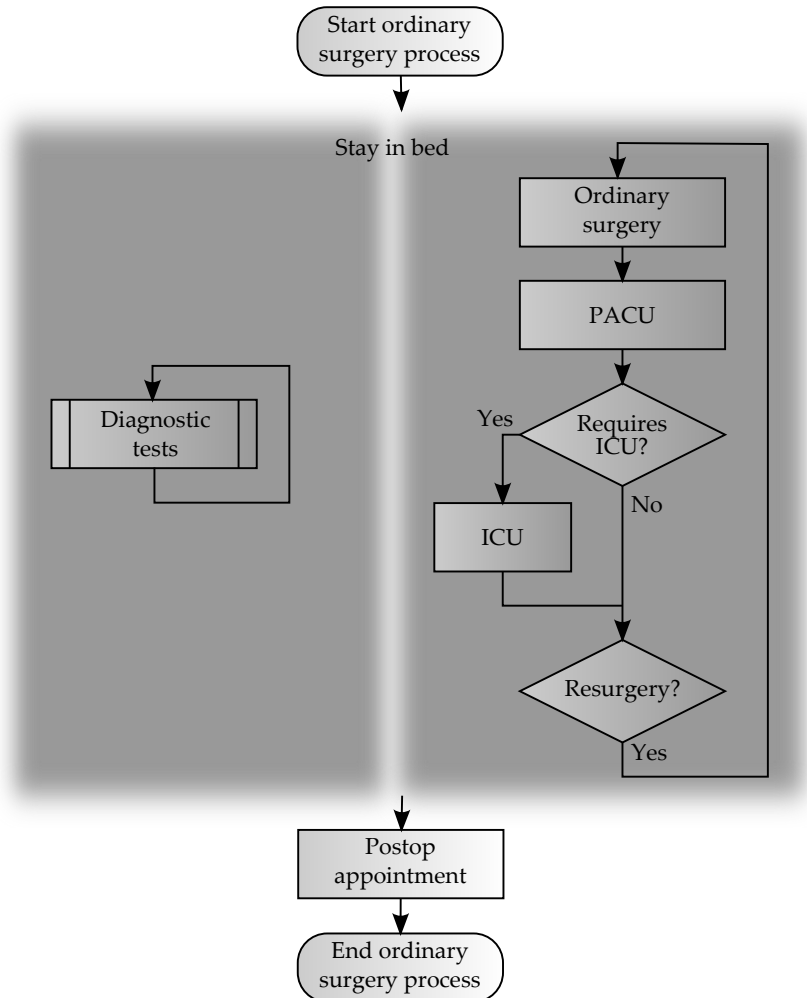


Figure 5.12. Flow for an ordinary surgical patient

This broad approach to surgery procedures may be refined by taking into account pathologies and specific surgical procedures. However, for the sake of simplicity, we will always define a typical surgical team as consisting of a surgeon, a scrub nurse, an anaesthetist and a circulating nurse. Anaesthetists are shared resources, whereas the remaining staff is normally attached to a department.

5.6.3. Central Services

Diagnostic tests, which appear in most flowcharts, are requested from central services and may include X-ray, USS, laboratory tests, etc., as shown in Figure 5.13.

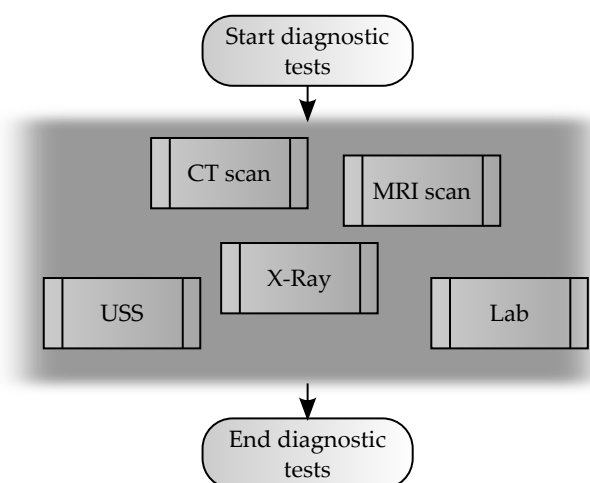


Figure 5.13. Diagnostic tests

While each test involves specific procedures and personnel, at a very high level of abstraction, the activity can be reduced to the test itself, that is, taking an X-ray, blood sample, etc., and the analysis of the test results, summarised in a report. Figure 5.14 presents this simplified schema. Specialised staff (technicians, nurses or doctors depending on the test) are in charge of those steps.

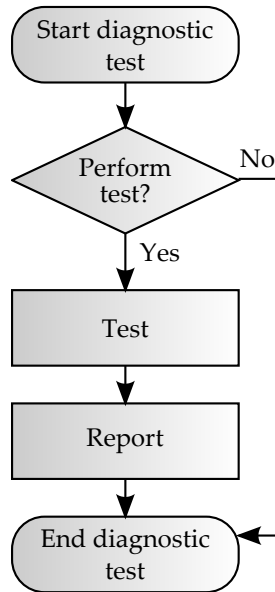


Figure 5.14. Standard diagnostic test

It would certainly be possible to adjust the level of detail of the different services. For example, some additional stages could be added to laboratory tests, as shown in Figure 5.15. This schema represents the typical flowchart for a hospital with a central laboratory in charge of taking and preparing the samples. Nurses take the samples, whereas lab technicians and specialised nurses carry out the tests.

5.7. Computational Model

The computational model developed with PSIGHOS is intended to fulfil the objectives previously introduced in Section 5.5. The use of Java simplifies the adoption of a modular design, which establishes a basis for handling the scalability of the model, and for future enhancements. The main classes are summarised in Table 5.3.

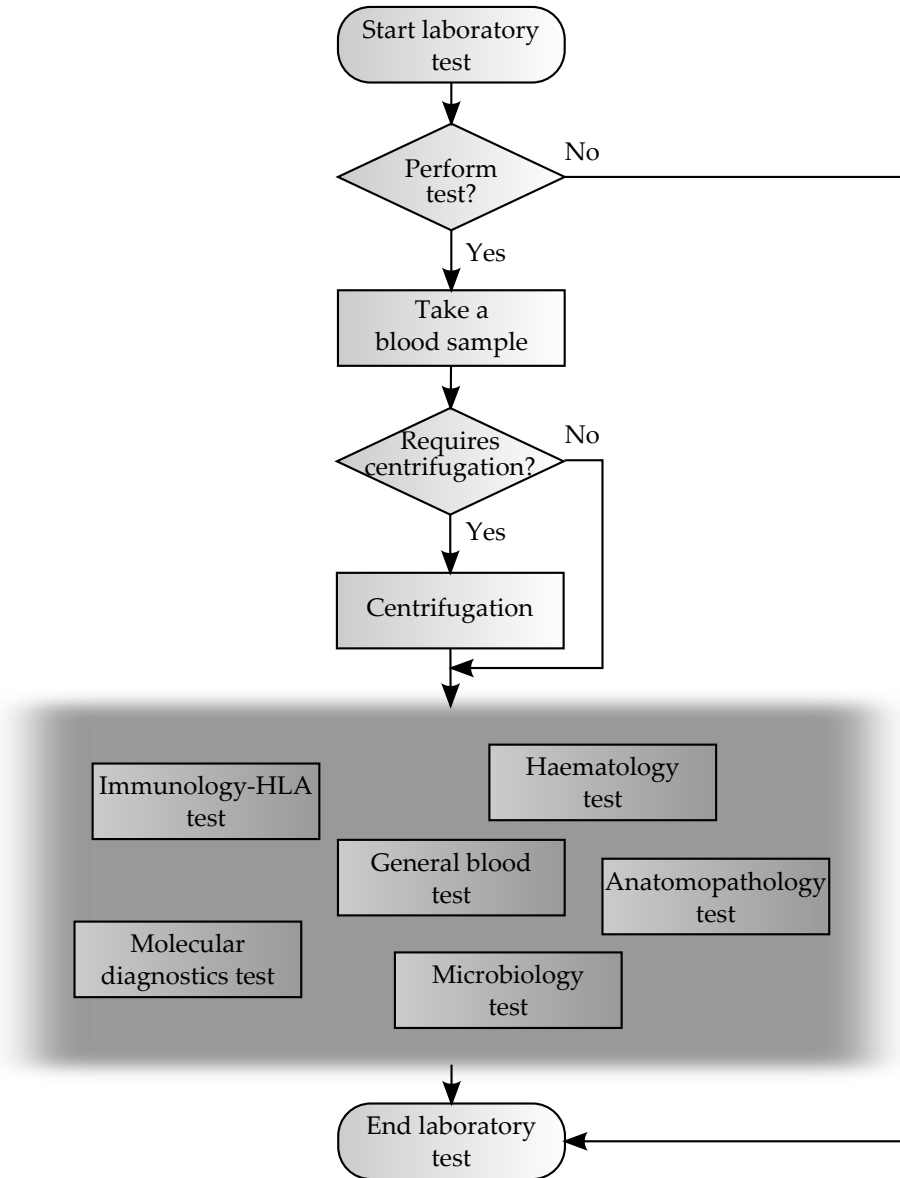


Figure 5.15. Laboratory tests

Table 5.3. Main classes of the hospital model with SIGHOS

Class	Description
HospitalModel	The core class for building the whole model. The remaining classes for defining different submodels are invoked within this class.
CentralServicesSubModel	This class describes the resources, activities and workflows involving the hospital's central services.
CentralLabModel	This class describes the resources, activities and workflows involving the hospital's central laboratories. The level of detail of the central laboratories, as already introduced in the conceptual model, is somewhat higher than that of the remaining central services.
StdMedicalDptModel	This class describes the resources, activities and workflows involving the average hospital MD. Instead of creating a unique module for the hospital's MDs, a single department is modelled. HospitalModel invokes as many instances of this class as there are MDs in the hospital, and parameterises each instance to meet the characteristics of a specific department.
StdSurgicalDptModel	This class is the SD equivalent of the StdMedicalDptModel class.
SurgicalDptSharedModel	This class incorporates the resources and activities that are shared by all the SDs. A similar class could be created for MDs if needed.

Though adaptable to user needs, PSIGHOS encourages a methodological approach to develop a computational model. Hence, the construction of a model can be accomplished by adhering to a set of ordered steps:

1. Identifying the different `Element` Types that appear in the system
2. Identifying `Resource` Types
3. Identifying individual `Resources` and defining timetables by associating resource scheduling with roles
4. Identifying typical `Workgroups` required to perform tasks in the system
5. Identifying `Activities`
6. Associating `Workgroups` and durations with each time-driven `Activity`
7. Defining the `Workflows` executed by the `Elements` in the system
8. Associating `Workgroups` and `Workflows` with each flow-driven `Activity`
9. Associating `Element` Types with `Workflows` and with creation patterns by means of `Element Creators` and `Generators`.

The following sections will describe each submodel in Table 5.3 by focusing on the identification of the modelling components and on the user-customisable parameters.

5.7.1. Some Basic Features of the Computational Model

Before describing each specific submodel, it is advisable to establish some shared features and basic characteristics of the computational model.

First, the time unit of the simulation is the *minute*. Using a higher temporal resolution does not provide any advantage, nor does it increase the accuracy of the simulation. Actually, according to the data analysis performed on the information systems of the HUNSC (see Subsection 5.3.1), a 5-minute resolution

is more suitable for most activities. Hence, we will attempt to adhere to this last resolution as much as possible.

The computational model comprises both human and material resources. With respect to material resources, they are considered to be continuously available (no failures, shutdowns or any other periods of unavailability are taken into account). Human resources (save for a few exceptions) start their work day at 8 am and work eight hours a day from Monday to Friday.

Since the model involves only elective patients, their arrival pattern has been simplified. There is a regular influx of patients who arrive daily at 7:50 am, the rate depending on the service and the type of patient. By using this arrival pattern, we avoid the need to design a strategy for scheduling referred patients. Obviously, if the objectives of the simulation model were different, this could be easily changed.

The distinction between medical and SDs can be considered too broad, since many departments do not perfectly fit into either of these categories. The key is to keep the model as simple and general as possible. However, if a certain department required a more detailed model, adding another class and connecting it to the schema designed would be trivial.

PSIGHOS utilises the *factory* design pattern (Gamma et al., 1995) to instantiate simulation objects. The factory pattern defines an interface for creating objects, instead of providing a direct access to the class constructor. The reasons that support the use of this pattern are twofold. On the one hand, the creation of new simulation objects is homogenised, and some internal implementation details, such as assigning a unique identifier to the object, are hidden from the user of the library. On the other hand, PSIGHOS allows a user to incorporate customised code to the simulation objects by means of hooks. The code is introduced as a string and must be dynamically compiled. Thus, the factory hides the dynamic compilation mechanism. Listing 5.1 shows some methods included in a factory of simulation objects.

Listing 5.1. A factory of simulation objects

```

public interface SimulationObjectFactory {
    Simulation getSimulation();
    ResourceType getResourceTypeInstance(String description) throws
        ClassCastException;
    ResourceType getResourceTypeInstance(String description,
        SimulationUserCode userMethods) throws ClassCastException;
    ...
}

```

The complete PSIGHOS Application Programming Interface (API) is described in Appendix B.

5.7.2. Central Services

The description of the central services in the conceptual model is very simple. Hence, the translation into a computational model is almost trivial.

Table 5.4 enumerates the simulation components defined for the central services. This prototype includes only two of the central services, USS and X-rays, but adding extra services would be straightforward.

Table 5.4. Simulation components for central services

Resource Types	USS Technician Radiology Technician
Resources	USS Technicians Radiology Technicians
Workgroups	{<USS Technician, 1> {<Radiology Technician, 1>
Time-driven Activities	USS Test Radiology Test USS Report Radiology Report

Table 5.5 summarises the parameters that can be modified to customise this submodel. In this case, a user can change the number of technicians and the duration of the activities.

Table 5.5. Parameters for central services

Parameter	Description
<i>NTECHUSS</i>	Resources available for Nuclear Medicine
<i>LENGTH_USSTEST</i>	Duration of the test
<i>LENGTH_USSREPORT</i>	Duration of the preparing the results report
<i>NTECHRAD</i>	Resources available for Radiology
<i>LENGTH_RADTEST</i>	Duration of the test
<i>LENGTH_RADREPORT</i>	Duration of preparing the results report

The code required to create the simulation components listed above invokes the corresponding methods of the factory. The example in Listing 5.2 illustrates the final implementation of the central services.

Listing 5.2. Model of the USS service with SIGHOS

```
// Creates the resource type
ResourceType rtUSS = factory.getResourceTypeInstance("USS Technician");

// Creates the resources
for (int i = 0; i < NTECHUSS; i++) {
    Resource res = factory.getResourceInstance("USS Technician " + i);
    // Associates resources with timetables and resource types
    // Methods getStdHumanResource... return standard values
    res.addTimeTableEntry(getStdHumanResourceCycle(),
        getStdHumanResourceAvailability(), rtUSS);
}

// Creates the workgroup
WorkGroup wgUSS = factory.getWorkGroupInstance(new ResourceType[]
    {rtUSS}, new int[] {1});
```

```
// Creates the activities
// The test has priority 2 and none of the possible modifiers
TimeDrivenActivity actUSSTest =
    factory.getTimeDrivenActivityInstance("USS Test", 2,
EnumSet.noneOf(TimeDrivenActivity.Modifier.class));
// The report has the same priority but does not require the patient
// to be present
TimeDrivenActivity actUSSReport =
    factory.getTimeDrivenActivityInstance("USS Report", 2,
EnumSet.of(TimeDrivenActivity.Modifier.NONPRESENTIAL));

// Associates the activities with the workgroups, and sets their
// duration
actUSSTest.addWorkGroup(LENGTH_USSTEST, wgUSS);
actUSSReport.addWorkGroup(LENGTH_USSREPORT, wgUSS);
```

This submodel does not comprise new `Element` Types but creates the simple workflows already depicted in Figure 5.14. Listing 5.3 shows the code required to create the workflow for the USS diagnostic test.

Listing 5.3. Model for the workflow of the USS service with `SIGHOS`

```
SingleFlow test = (SingleFlow) factory.getFlowInstance("SingleFlow",
actUSSTest);
SingleFlow report = (SingleFlow) factory.getFlowInstance("SingleFlow",
actUSSReport);
test.link(report);
```

5.7.3. Central Laboratories

According to the conceptual model, laboratory tests are more thoroughly described than the other central services. As listed in Table 5.6, more specialised roles and activities are taken into account.

Some of the `Resources` require additional clarification. For example, *24/7 Lab Technicians* have the role of *Lab Technicians* but work 24 hours a day, 7 days a week. While no worker with such a schedule exists, each of these `Resources`

comprises three *real* shiftwork resources. In a more realistic hospital model, where measures of employee workload are significant, the actual employee shifts of the workers would be utilised. However, from the point of view of this simulation study, this approach leads us to the same results while at the same time considerably simplifying the model.

Table 5.6. Simulation components for central laboratories

Resource Types	Lab Technician
	Test slot
	Centrifugation slot
	Lab Nurse
	Lab Specialist Nurse
	Haematology Lab Technician
	Haematology Lab Nurse
	Haematology Lab Test slot
	Microbiology Lab Technician
	Microbiology Lab Nurse
	Microbiology Lab Test slot
	Anatomopathology Lab Technician
	Anatomopathology Lab Nurse
	Anatomopathology Lab Test slot
Resources	Lab Technicians
	24/7 Lab Technicians
	Test slots
	Centrifugation slots
	Lab Nurses
	Lab Specialist Nurses
	Haematology Lab Technicians

Table 5.6. Simulation components for for central laboratories – Continued

Resources	Haematology Lab Nurses Haematology Lab Test slots Microbiology Lab Technicians Microbiology Lab Nurses Microbiology Lab Test slots Anatomopathology Lab Technicians Anatomopathology Lab Nurses Anatomopathology Lab Test slots
Workgroups	{<Lab Nurse, 1>} {<Centrifugation slot, 1>} {<Test slot, 1>, <Lab Technician, 1>} {<Test slot, 1>, <Lab Specialist Nurse, 1>} {<Haematology Lab Test slot, 1>, <Haematology Lab Technician, 1>} {<Haematology Lab Test slot, 1>, <Haematology Lab Nurse, 1>} {<Microbiology Lab Test slot, 1>, <Microbiology Lab Technician, 1>} {<Microbiology Lab Test slot, 1>, <Microbiology Lab Nurse, 1>} {<Anatomopathology Lab Test slot, 1>, <Anatomopathology Lab Technician, 1>} {<Anatomopathology Lab Test slot, 1>, <Anatomopathology Lab Nurse, 1>}
Time-driven Activities	Take a sample OP Centrifugation OP Test OP Haematology Test OP Microbiology Test OP Anatomopathology Test OP

Table 5.6. Simulation components for for central laboratories – Continued

Time-driven Activities	Take a sample IP
	Centrifugation IP
	Test IP
	Haematology Test IP
	Microbiology Test IP
	Anatomopathology Test IP

An example of Resources having multiple roles are *Specialist* and *Haematology Nurses*, who can also play the role of regular *Lab Nurses*. Hence, they not only perform specific tests, but can take blood samples if required. This can be done by simply associating an additional timetable entry with a Resource, as Listing 5.4 illustrates.

Listing 5.4. Model for central lab specialist nurses with SIGHOS

```
// Creates the resource types
ResourceType rtNurse = factory.getResourceTypeInstance("Lab Nurse");
ResourceType rtXNurse = factory.getResourceTypeInstance("Lab
    Specialist Nurse");

// Creates the resources
for (int i = 0; i < NXNURSES; i++) {
    Resource res = factory.getResourceInstance("Lab Specialist Nurse "
        + i);

    // We simply associate two timetable entries with each resource
    res.addTimeTableEntry(getStdHumanResourceCycle(),
        getStdHumanResourceAvailability(), rtNurse);
    res.addTimeTableEntry(getStdHumanResourceCycle(),
        getStdHumanResourceAvailability(), rtXNurse);
}
```

The notion of *slot* demands further explanation too. Laboratories carry out a wide variety of tests, both manual and automated. It is beyond the scope of

this approach to present a detailed and comprehensive model for the different tests. Instead, our model dramatically simplifies this schema and defines a test as an *Activity* that requires one *Specialised Nurse* or *Technician* and some kind of *machine slot*. A *slot* is an abstraction that represents the “portion” of a machine required by a member of the staff to carry out a single test.

Table 5.6 shows the same *Activities* twice, the only difference being the suffix: *OP* and *IP*, which stand for *OutPatient* and *InPatient* *Activities*. Indeed, the homologous *Activities* use the same *Workgroup* and take the same time. The difference is the priority: inpatients are generally attended to with a higher priority than outpatients. Hence, a higher priority value is assigned to *IP* *Activities* upon creation. This approach may be easily extended to emergency or ICU patients, who require an even higher priority.

As noted earlier, our model allows any test to be performed either by a *Specialised Technician* or a *Specialised Nurse*. Listing 5.5 exemplifies this.

Listing 5.5. Model for a central lab laboratory test with SIGHOS

```
ResourceType rtSlot = factory.getResourceTypeInstance("Test Slot");
ResourceType rtTech = factory.getResourceTypeInstance("Lab
    Technician");
...
WorkGroup wgTest1 = factory.getWorkGroupInstance(new ResourceType[]
    {rtSlot, rtTech}, new int[] {1, 1});
WorkGroup wgTest2 = factory.getWorkGroupInstance(new ResourceType[]
    {rtSlot, rtXNurse}, new int[] {1, 1});

// A priority 2 non-presential activity is created for the OP
// laboratory test
TimeDrivenActivity actOutTest =
    factory.getTimeDrivenActivityInstance("Test OP", 2,
    EnumSet.of(TimeDrivenActivity.Modifier.NONPRESENTIAL));

// According to the priorities, Technicians are preferable to
// Specialised Nurses
actOutTest.addWorkGroup(LENGTH_TEST, 0, wgTest1);
actOutTest.addWorkGroup(LENGTH_TEST, 1, wgTest2);
```

The usage of priorities in the Workgroups involves a preference of the simulation engine for *Technicians* over *Specialised Nurses*; that is, only if no *Technicians* are available will a *Specialised Nurse* be assigned. Priorities here try to prevent the monopolisation of *Specialised Nurses* by this Activity, since they may be required in different tasks.

Table 5.7 lists the parameters that a user can modify to affect the behaviour of the central laboratories. In essence, the number of Resources of each type and the length of each Activity can be customised.

Table 5.7. Parameters for central laboratories

Parameter	Description
<i>NTECH</i>	Number of technicians
<i>N24HTECH</i>	Number of technicians working 24 hours
<i>NNURSES</i>	Number of nurses
<i>NXNURSES</i>	Number of specialist nurses
<i>NSLOTS</i>	Test “Slots”
<i>NCENT</i>	Centrifugation “Slots”
<i>LENGTH_SAMPLE</i>	Time required to take a sample
<i>LENGTH_CENT</i>	Time required to centrifuge
<i>LENGTH_TEST</i>	Duration of test slot
<i>NHAETECH</i>	Number of Haematology Lab technicians
<i>NHAENURSES</i>	Number of Haematology Lab nurses
<i>NHAESLOTS</i>	“Slots” for Haematology Lab tests
<i>LENGTH_HAETEST</i>	Duration of Haematology Lab test slot
<i>NMICROTECH</i>	Number of Microbiology Lab technicians
<i>NMICRONURSES</i>	Number of Microbiology Lab nurses
<i>NMICROSLOTS</i>	“Slots” for Microbiology Lab tests
<i>LENGTH_MICROTEST</i>	Duration of Microbiology Lab test slot
<i>NPATTECH</i>	Number of Anatomopathology Lab technicians

Table 5.7. Parameters for central laboratories – Continued

Parameter	Description
<i>NPATNURSES</i>	Number of Anatomopathology Lab nurses
<i>NPATSLOTS</i>	“Slots” for Anatomopathology Lab tests
<i>LENGTH_PATTEST</i>	Duration of Anatomopathology Lab test slot

Neither the central laboratory nor central services submodel defines *Element Types*, though they do define the *Workflows* schematised in Figure 5.15. Laboratory tests are performed based on a probability, which depends on the specific department requesting the tests.

5.7.4. Medical Departments

Table 5.8 summarises the main components required to create the model of a medical department.

Table 5.8. Simulation components for medical departments

Element Types	Elective Patient Chronic Patient
Resource Types	Doctor (First Appointment) Doctor (Follow-up Appointment) Bed
Resources	Doctors Beds
Workgroups	{<Doctor (First Appointment), 1>} {<Doctor (Follow-up Appointment), 1>} {<Bed, 1>}
Time-driven Activities	First OP Appointment Follow-up OP Appointment

Table 5.8. Simulation components for for medical departments – Continued

Time-driven Activities	Waiting for Next Appointment
	Waiting for Admission
	Waiting for Recovery
	Waiting for Next IP Test
Flow-driven Activities	Ward Stay

Two different types of patients appear, since they involve different decisions in the workflow. *Chronic Patients*, due to their pathologies, are always present in the system. An enhanced model should include some triggers that may lead these patients to leave the hospital, such as *exitus*, moving house or simply a personal decision. *Elective Patients* on the other hand, have a finite number of appointments, and are eventually admitted or simply discharged if no inpatient medical treatment is required.

Doctors split their timetable into first and follow-up appointments. A real hospital utilises more complex strategies to decide which doctor attends to a specific patient, based on characteristics of the patient (first or follow-up appointment, pathology...) and the doctor's experience. These strategies could be added to the model by incorporating new roles for the doctors, and new `Element` Types resulting in a finer division of patients according to their characteristics.

Although surgeries seem to be missing from the model, they are in fact *aggregated* to the notion of doctor. Aggregation is a technique to reduce complexity that involves the implicit representation of some parts of the model in others. Therefore, the availability of doctors is regarded as implying the readiness of surgeries.

A final remark is required regarding the `Activities` defined within this sub-model. The workflow of a medical patient includes some *wait periods*. For example, a certain time must pass between successive appointments. *Waits* are included in the patient's workflow with PSIGHOS by means of `Activities` that

do not require any Resource. Listing 5.6 describes the creation of one such Activities.

Listing 5.6. Model of a “waiting” activity with SIGHOS

```
// First, a “dummy” WG is created
WorkGroup dummyWG = factory.getWorkGroupInstance(new ResourceType[]
    {}, new int[] {});

// A non-presential activity with maximum priority
TimeDrivenActivity act =
    factory.getTimeDrivenActivityInstance("Waiting for next App.", 0,
        EnumSet.of(TimeDrivenActivity.Modifier.NONPRESENTIAL));

// The length of the wait is assigned
act.addWorkGroup(LENGTH_OP2OP, dummyWG);
```

Ward Stay is a Flow-Driven Activity, that is, its duration does not depend directly on time, but on the execution of an inner workflow. Specifically, a patient stays in the ward for a certain number of days (*Waiting for Recovery*) but, at the same time, a set of diagnostic tests and treatments may be performed. Listing 5.7 presents a sketch of the code required to create a Flow-Driven Activity.

Listing 5.7. Model for a flow-driven activity within SIGHOS

```
// A “stay in bed” single flow is created with the Waiting for
Recovery activity
SingleFlow stayInBed =
    (SingleFlow)factory.getFlowInstance("SingleFlow",
        actWaitingRecovery);

// An interleaved routing flow (WCP40) invokes...
InterleavedRoutingFlow parallelStay = (InterleavedRoutingFlow)
    factory.getFlowInstance("InterleavedRoutingFlow");
// ...the previously defined activity...
parallelStay.addBranch(stayInBed);
// ...and a method that builds a workflow of diagnostic tests
parallelStay.addBranch(getDiagnosticTests());
```

```
// The flow-driven activity is created
FlowDrivenActivity actStay = (FlowDrivenActivity)
    factory.getFlowDrivenActivityInstance("Ward Stay");
// The workgroup wgBed <BED, 1> is used
actStay.addWorkGroup(parallelStay, wgBed);
```

Table 5.9 lists the parameters of this submodel. The number of *resources* as well as the length of the *activities* are shown as usual. As mentioned in the previous section, every department defines the probabilities of performing each kind of test by means of the *PROB_XXX_OP* and *PROB_XXX_IP* parameters.

Table 5.9. Parameters for medical departments

Parameter	Description
<i>NDOCTORS</i>	Doctors available
<i>NBEDS</i>	Beds available
<i>LENGTH_OP1</i>	Length of First appointment
<i>LENGTH_OP2</i>	Length of successive appointment
<i>LENGTH_OP2OP</i>	Time between successive appointments
<i>LENGTH_OP2ADM</i>	Time between last appointment and admission
<i>PROB_RAD_OP</i>	Probability of performing an X-Ray test during appointments
<i>PROB_NUC_OP</i>	Probability of performing a scanner test during appointments
<i>PROB_LAB_OP</i>	Probability of performing a lab test during appointments
<i>PROB_LABCENT_OP</i>	Probability of centrifugation of sample during appointments
<i>PROB_LABLAB_OP</i>	Probability of performing a central lab test during appointments
<i>PROB_LABHAE_OP</i>	Probability of performing a Haematology lab test during appointments

Table 5.9. Parameters for medical departments – Continued

Parameter	Description
<i>PROB_LABMIC_OP</i>	Probability of performing a Microbiology lab test during appointments
<i>PROB_LABPAT_OP</i>	Probability of performing an Anatomopathology lab test during appointments
<i>PROB_RAD_IP</i>	Probability of performing an X-Ray test during stay
<i>PROB_NUC_IP</i>	Probability of performing a scanner test during stay
<i>PROB_LAB_IP</i>	Probability of performing a lab test during stay
<i>PROB_LABCENT_IP</i>	Probability of centrifugation of sample during stay
<i>PROB_LABLAB_IP</i>	Probability of performing a central lab test during stay
<i>PROB_LABHAE_IP</i>	Probability of performing a Haematology lab test during stay
<i>PROB_LABMIC_IP</i>	Probability of performing a Microbiology lab test during stay
<i>PROB_LABPAT_IP</i>	Probability of performing an Anatomopathology lab test during stay
<i>LOS</i>	Length of stay after admission
<i>PROB_ADM</i>	Probability of being admitted
<i>NCPATIENTS</i>	Number of chronic patients that arrive at the department
<i>NPATIENTS</i>	Number of patients that arrive at the department
<i>INTERARRIVAL</i>	Time between successive patient arrivals
<i>ITERSUCC</i>	Number of successive appointments
<i>PROB_1ST_APP</i>	Probability of a doctor being assigned to first appointment
<i>HOURS_INTERIPTEST</i>	Time (in hours) between successive IP tests

5.7.5. Surgical Departments

Since the developed model includes some facilities and resources shared among the different SDs, two classes have been declared: *SurgicalDptSharedModel* defines the shared structures, whereas *StdSurgicalDptModel* represents a single parameterised surgical department model.

The shared facilities and resources defined in the model are the *PACU*, the *ICU* and the *Anaesthetists*, as shown in Table 5.10.

Table 5.10. Simulation components shared by surgical departments

Resource Types	PACU Bed ICU Bed Anaesthetist
Resources	PACU Beds ICU Beds Anaesthetists
Workgroups	{<PACU Bed, 1> {<ICU Bed, 1>
Time-driven Activities	PACU stay ICU stay

The number of Resources can be set by means of the parameters listed in Table 5.11.

Table 5.11. Parameters for shared surgical departments resources

Parameter	Description
<i>NBEDS_PACU</i>	Number of beds available in P.A.C.U.
<i>NBEDS_ICU</i>	Number of beds available in I.C.U.
<i>NANAESTHETISTS</i>	Number of anaesthetists

SDs make use of the shared resources but also define their own modelling components, similar to those of MDs. Table 5.12 shows the contents of the SD computational model.

Table 5.12. Simulation components for surgical departments

Element Types	Surgical Patient Short-stay Patient Ambulatory Patient
Resource Types	Bed Short-stay Bed Doctor (First Appointment) Doctor (Follow-up Appointment) Surgeon Operating Theatre Scrub Nurse Circulating Nurse
Resources	Beds Short-stay Beds Doctors Surgeons Operating Theatres Scrub Nurses Circulating Nurses
Workgroups	{<Doctor (First Appointment), 1>} {<Doctor (Follow-up Appointment), 1>} {<Short-stay Bed, 1>} {<Bed, 1>}

Table 5.12. Simulation components for for surgical departments – Continued

Workgroups	{<Operating Theatre, 1>, <Surgeon, 1>, <Scrub Nurse, 1>, <Anaesthetist, 1>, <Circulating Nurse, 1>}
Time-driven Activities	First OP Appointment
	Follow-up OP Appointment
	Waiting for Next Appointment
	Waiting for Post-Surgery Appointment
	Waiting for Admission
	Waiting for Next IP Test
	Post-Surgery Appointment
	Surgery
	Surgery (Short-stay)
	Ambulatory Surgery
Recovering after Surgery	
Recovering after Surgery (Short-stay)	
Flow-driven Activities	Ward Stay
	Ward Short-stay

As derived from the conceptual model described in Subsection 5.6.2, surgical departments comprise three patient types, depending on the characteristics of the surgical procedure they require: ambulatory, short-stay and ordinary. Three different activities, one per element type, represent a surgery in the model. Though a different duration can be assigned to each activity, all three utilise the same workgroup (*OT + Surgeon + Scrub Nurse + Anaesthetist + Circulating Nurse*). A more detailed model would include a different surgery type per pathology and more flexible workgroups.

All the members of a surgical team except for *Circulating Nurses* are associated with a single surgery. *Circulating Nurses* are a special resource that is not bound to a specific surgery, but is rather shared between two and even more OTs per-

forming simultaneous operations. A simple and straightforward solution for this problem is stated by Günel and Pidd (2006). They propose the use of a *Multitasking factor* M to fragment each resource into M parts. Each part is considered a *mini Circulating Nurse* and can be fully utilised by any activity requiring it. The main drawback of this approach may be an underestimation of the resource usage. However, since wait times and patient flows are our main concern, the use of a *mini Circulating Nurse* can be regarded as a suitable simplification.

The parameters available to customise a SD are summarised in Table 5.13.

Table 5.13. Parameters for surgical departments

Parameter	Description
<i>NBEDS</i>	Beds available
<i>NSBEDS</i>	Beds available for short stays
<i>NOPTHEATRES</i>	Operating Theatres available for the department
<i>NSURGEONS</i>	Surgeons available for the department
<i>NDOCTORS</i>	Doctors available for the department
<i>NSCRUBNURSES</i>	Scrub Nurses available for the department
<i>NCIRCNUMSES</i>	Circulating Nurses available for the department
<i>PROB_RAD_OP</i>	Probability of performing an X-Ray test during appointments
<i>PROB_NUC_OP</i>	Probability of performing a scanner test during appointments
<i>PROB_LAB_OP</i>	Probability of performing a lab test during appointments
<i>PROB_LABCENT_OP</i>	Probability of centrifugation of sample during appointments
<i>PROB_LABLAB_OP</i>	Probability of performing a central lab test during appointments
<i>PROB_LABHAE_OP</i>	Probability of performing a Haematology lab test during appointments

Table 5.13. Parameters for surgical departments – Continued

Parameter	Description
<i>PROB_LABMIC_OP</i>	Probability of performing a Microbiology lab test during appointments
<i>PROB_LABPAT_OP</i>	Probability of performing an Anatomopathology lab test during appointments
<i>PROB_RAD_IP</i>	Probability of performing an X-Ray test during stay
<i>PROB_NUC_IP</i>	Probability of performing a scanner test during stay
<i>PROB_LAB_IP</i>	Probability of performing a lab test during stay
<i>PROB_LABCENT_IP</i>	Probability of centrifugation of sample during stay
<i>PROB_LABLAB_IP</i>	Probability of performing a central lab test during stay
<i>PROB_LABHAE_IP</i>	Probability of performing a Haematology lab test during stay
<i>PROB_LABMIC_IP</i>	Probability of performing a Microbiology lab test during stay
<i>PROB_LABPAT_IP</i>	Probability of performing an Anatomopathology lab test during stay
<i>LENGTH_OP1</i>	Length of First appointment
<i>LENGTH_OP2</i>	Length of successive appointment
<i>LENGTH_POP</i>	Length of post-surgery appointment
<i>LENGTH_OP2ADM</i>	Time between last appointment and admission
<i>LENGTH_OP2OP</i>	Time between successive appointments
<i>LENGTH_SUR2POP</i>	Time between surgery and post-surgery appointment
<i>LENGTH_SUR</i>	Length of surgery
<i>LENGTH_SSUR</i>	Length of surgery (short-stay)
<i>LENGTH_ASUR</i>	Length of ambulatory surgery
<i>LENGTH_SPACU</i>	Length of P.A.C.U. stay after surgery (short-stay)
<i>LENGTH_APACU</i>	Length of P.A.C.U. stay after ambulatory surgery
<i>LENGTH_PACU</i>	Length of P.A.C.U. stay

Table 5.13. Parameters for surgical departments – Continued

Parameter	Description
<i>LENGTH_ICU</i>	Length of I.C.U. stay
<i>LENGTH_SUR2EXIT</i>	Minimum stay after surgery
<i>LENGTH_SSUR2EXIT</i>	Minimum stay after surgery (short-stay)
<i>PROB_ADM</i>	Probability of being admitted
<i>NPATIENTS</i>	Number of patients that arrive at the department
<i>INTERARRIVAL</i>	Time between successive patient arrivals
<i>NAPATIENTS</i>	Number of patients that arrive at the department for ambulatory surgery
<i>AINTERARRIVAL</i>	Time between successive patient arrivals for ambulatory surgery
<i>NSPATIENTS</i>	Number of patients that arrive at the department for surgery (short-stay)
<i>SINTERARRIVAL</i>	Time between successive patient arrivals for surgery (short-stay)
<i>ITERSUCC</i>	Number of successive appointments
<i>PROB_1ST_APP</i>	Probability of a doctor being assigned to first appointment
<i>HOURS_INTERIPTEST</i>	Time (in hours) between successive IP tests

5.8. Results

The computational model described in the previous section, though far from being considered realistic, is a highly parameterisable and scalable example.

Initially, the parameters for defining the hospital's central services and central laboratories are defined. Clearly, the model for central services is too simple and cannot be parameterised to resemble the real system. We have, however, tried to adhere as much as possible to the characteristics of the HUC's central labo-

ratories by using the real number of nurses, technicians and other resources, as shown in Table 5.14. Still, the model is only an approximation since no resources for emergencies have been considered.

Table 5.14. Resources of the HUC's central laboratories

Parameter	Value
Technicians	23
24/7 Technicians	5
Nurses	16
Specialist nurses	10
"Slots" for tests	150
"Slots" for centrifugation	160
Haematology technicians	2
Haematology Nurses	5
"Slots" for Haematology tests	40
Microbiology Technicians	10
Microbiology Nurses	0
"Slots" for Microbiology tests	50
Anatomopathology Technicians	6
Anatomopathology Nurses	1
"Slots" for Anatomopathology tests	50

We have created three MD and three SD *templates*. Each template is intended to highlight certain characteristics of a department. Hence, a traumatology-like SD is created that makes extensive use of X-rays; or an ophthalmology-like MD that only requests a few clinical tests in the central laboratories.

Having fixed the parameters of the central services and central laboratories submodels, the different testing scenarios are defined based on the number of *departments per template* (hereafter DxT) included in the simulation model. Three

scenarios are utilised that simply vary the amount of DxT : 1, 2 and 4, which results in scenarios with 6, 12 and 24 departments.

As explained in Section 4.3, concurrent access to simulation Resources can affect the performance of PSIGHOS. Two main structures were introduced to solve this problem: *Activity Managers (AMs)* and dynamic *Conflict Zones (CZs)*. The former creates a static partition of the model based on the Resource Types and Activities, whereas the latter makes use of a dynamically built “zone” where conflicts between specific Resources simultaneously requested by several elements can be solved. As already shown with the test benchmark presented in Section 4.4, PSIGHOS yields more opportunities to exploit the parallelism of a model as the number of AMs increases, and the number of Resources with overlapped timetable entries decreases. Hence, we can analyse these characteristics of the computational model prior to its execution in order to enhance its potential parallelism. Table 5.15 summarises the number of Resource Types and Activities created within the hospital model.

Table 5.15. Resource types and activities in the computational model

Submodel	Resource Types	Activities
Central Services	2	4
Central Laboratories	14	12
Surgical Departments (shared)	3	2
Surgical Department	8	14
Medical Department	3	7

The Activities and Resource Types defined within the central services/laboratories are shared among all the surgical and medical departments. Hence, there are 19 shared Resource Types and 18 shared Activities. The number of Resource Types for the model $|RT|$ can be computed by using the formula below. Also, let $|MD|$ and $|SD|$ be the number of medical and surgical departments, respectively. $|RT|$ can be computed as follows:

$$|RT| = 19 + 3 * |MD| + 8 * |SD|$$

where

$|MD|$ is the number of medical departments. Since there are three MD templates, $|MD| = DxT * 3$.

$|SD|$ is the number of surgical departments, with $|SD| = DxT * 3$.

Analogously, the number of Activities $|A|$ can be computed as

$$|A| = 18 + 7 * |MD| + 14 * |SD|$$

A careful analysis of the model's Activities and Resource Types led us to detect the following groups of AMs:

- Central services use the same Resource Type for both the test and the report Activities. Thus, each service included in the model, that is, X-rays and USS, defines its own AM.
- Central laboratories perform IP and OP Activities that vary in priority but not workgroup. Thus, each pair of IP-OP Activities is grouped into the same AM.
- Anaesthetists are shared among all the surgery Activities. Hence, they all are placed in the same AM
- Follow-up OP appointments and post-operative OP appointments for a SD are performed by the same Resource type, and are thus handled by the same AM.
- Each of the remaining Activities is handled by a different AM.

The number of AMs, $|AM|$, can be computed similarly to $|RT|$ and $|A|$.

$$|AM| = 10(\text{shared}) + 1(\text{surgery}) + 7 * |MD| + 10 * |SD|$$

Table 5.16 summarises the number of components for each scenario. Taking into account the number of cores in the test machine (16) or typical personal computers (2-8), the number of AMs seems sufficient to provide opportunities for exploiting parallelism.

Table 5.16. Number of RTs, As and AMs in each scenario

DxT	$ RT $	$ A $	$ AM $
1	52	81	62
2	85	144	113
4	151	270	215

With respect to resources with overlapping timetables, Haematology and Specialised nurses, as mentioned in Subsection 5.7.3 can also play the role of nurses. Thus, taking a sample and testing a different sample could try to use the same Resource, even though these Activities are handled by different AMs. Doctors also have two timetable entries (for first and follow-up OP appointments), but they are set to be active at non-overlapping time periods.

The static analysis above is useful for detecting problematic situations and establishing an upper bound to the parallelism that can be exploited from the model. Nevertheless, the simulation of the model will only profit from the potential parallelism if the Elements and their Workflows produce enough simultaneous events. Hence, only the execution of the simulation will offer accurate conclusions.

Executing these tests yielded the results summarised in Figure 5.16 and in Table 5.17. Figure 5.16 shows the average execution time for each scenario, whereas Table 5.17 shows the speedup of the parallel version over the sequential SIGHOS. Each scenario was run for 24 simulated months and executed 5 times on the platform previously introduced in Table 4.4. The replicas were tested with the sequential SIGHOS and the version of PSIGHOS that showed the best performance in the synthetic tests, that is, the Hybrid EME like-3-Phase Approach (see

Section 4.13). In this last case (PSIGHOS), 2, 4, 8 and 16 threads (*Event Executors*) were tried.

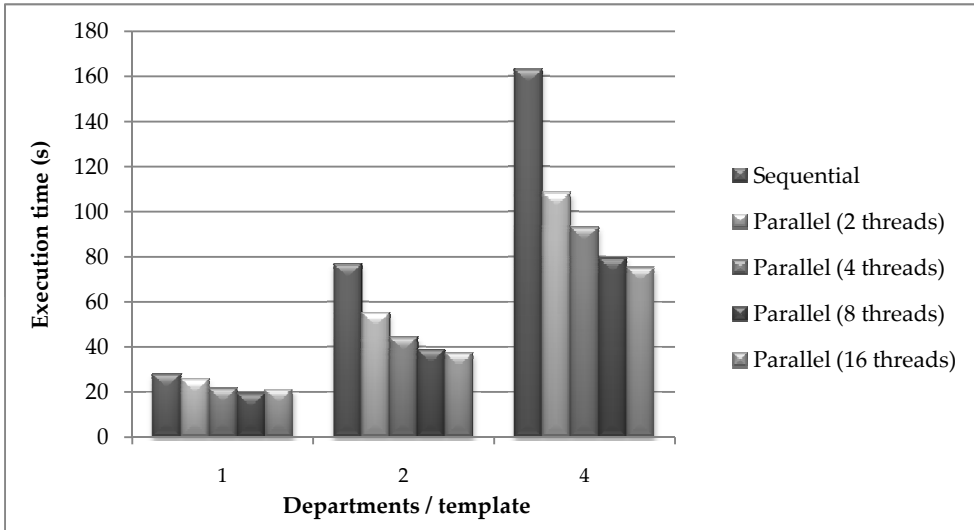


Figure 5.16. Execution time for different whole hospital model scenarios

Table 5.17. Speedup for different whole hospital model scenarios

Threads	DxT		
	1	2	4
2	1.085	1.395	1.505
4	1.301	1.739	1.753
8	1.457	1.984	2.055
16	1.317	2.075	2.174

Every $DxT = 1$ scenario exhibits low efficiency. Not only this, the speedup with 16 threads is even worse than that with 8 threads, probably due to this scenario's limited scale.

The scenarios with $DxT = 2$ and $DxT = 4$ present a similar behaviour. The efficiency with 2 threads is highly satisfactory, and is also adequate for 4 threads. However, adding more threads does not produce a significant increment in the speedup.

In spite of creating twice the events (around 14 million against 7 million), the gain of the scenario with $DxT = 4$ against that with $DxT = 2$ is barely noticeable. Clearly, adding a new department involves creating more independent activities that may be executed in parallel. However, since every department must access the central services/laboratories, requesting activities from these submodels bottlenecks the simulation.

5.9. Conclusions

In this chapter, a generic model of a whole hospital has been presented. The model is not intended to be an accurate representation of a hospital, but rather to illustrate the modelling capabilities of PSIGHOS and to test the performance of the simulator when faced with a practical case.

PSIGHOS' modelling capabilities were put to the test by capturing complex resource behaviours, such as multi-tasking, complex timetables and specialisation. A sophisticated treatment of the system entities, that is, the hospital patients, is also possible since several categories can be defined (thus opening the door to distinguishing between emergency and elective patients, and even based on their pathology), and different workflows can be associated with each. Furthermore, two types of activities were used: simple tasks, whose length is defined according to a predefined time function; and compound tasks, which are considered complete as an associated subprocess finishes.

With respect to the results, the simulation shows low scalability with the number of processors. Clearly, central services and laboratories, as well as the shared surgical resources (PACU, ICU and anaesthetists), bottleneck the execution of the model. Still, taking into consideration that the model is generic and has not

been especially adapted to exploit a parallel platform, its performance can be considered satisfactory when executing fewer than 4 threads.

Conclusions

The breakthrough in multi-core technology has given rise to a new scenario where parallelism is no longer part of the scientific playground, but a sensible choice for every application with high computational requirements. Business Process Simulation (BPS) is one of the fields that can profit from this technology, since large and complex models appear as the size of the organisation grows or the level of detail required to capture the system's semantics increases.

Traditionally, Parallel Discrete Event Simulation (PDES) has rarely been used beyond the scientific world due to a lack of generic solutions that can be applied to different models. Domain decomposition, which is the most popular PDES technique, requires a design that explicitly exploits the parallelism of a simulation model. This leads to ad-hoc solutions and requires a parallelism expert to develop the simulation.

Given this situation, this thesis has explored the feasibility of using multi-core computers to *automatically* and *efficiently* exploit parallelism for simulating Business Process (BPs) models.

The advances presented in this thesis can be grouped into three different areas:

1. *Generality*. Focusing as it has on the field of BPS, this thesis has not merely treated a specific case study or system. Instead, workflow patterns have been posed as a suitable formalism for generically describing the organisation's BPs. This thesis has offered several demonstrations to prove how the use of workflow patterns, along with the definition of high-level model-

ling structures adapted to the organisational language (such as activities, human resources or timetables), establish a well-suited simulation framework that allows a non-expert simulation user to quickly become familiar with the components required to create a model. A practical implementation in Java, called SIGHOS, has been presented that comprises all of these features.

2. *Automation.* Having established a suitable generic BPS framework, the next step was to allow the simulation engine to automatically exploit the parallelism present in the model being simulated by profiting from the resources of a multi-core computer. The automation concept implicitly involved hiding the implementation and handling of the parallel mechanisms from the simulation user. This thesis has presented a new algorithm that automatically distributes the simulation events to a set of Event Executors (EEs) by carefully managing the access to shared simulation resources. The algorithm combines concepts from the Three-Phase approach (see Subsection 1.2.4), the master-slave paradigm, and a parallel architecture with a centralised list and distributed events (see Subsection 1.3.5). PSIGHOS, a practical implementation of a parallel simulation tool using this algorithm, has been presented that preserves the modelling interface of the original SIGHOS.
3. *Efficiency.* BPS is a problem in which both resource contention and the communication among entities usually limit the degree to which the parallelism can be exploited. This thesis has analysed these problems and presented several advances in the static and dynamic management of shared resources so as to reduce resource contention. The experimental results have corroborated the efficiency of our proposal when faced with synthetic problems and shown encouraging speedups when tested using a generic, real-world model.

To the best of our knowledge, there is no similar approach in the field of BPS that combines these three features.

Contributions

Apart from the above advances, several research objectives are posed in the Introduction to this thesis. This section will present our contributions within each objective.

Identify the characteristics that make Discrete-Event System Simulation (DES) and, more specifically, process-oriented simulation a suitable approach for modelling and simulating BPs

Chapter 1 and Chapter 2 show that process-oriented simulation is the most widely used DES worldview in both free and commercial simulation tools. This stems from the use of high-level blocks to build models that hide the low-level implementation.

Then, Chapter 3 refines the main concepts used in BPS and highlights the importance of workflow patterns in obtaining a standard and usable definition of a BP. By carefully examining those definitions, mapping the BPS concepts onto those of process-oriented DES is a straightforward procedure. Based on those concepts, a Java BPS library called SIGHOS is presented.

Discuss the feasibility of automatically exploiting the parallelism in a BP model

Automated exploitation of parallelism is based on two main milestones: 1) the isolation of the model definition and the simulation execution and 2) the use of a parallel approach that focuses on the simulation engine and not on the specific model being simulated.

The first milestone is achieved by choosing an automated event translation for the low-level implementation of the process-oriented DES, as explained in

Section 3.4. Hence, the modelling interface is kept separate from the simulation engine.

The second milestone is analysed in Section 4.1. After conducting a survey of the most important techniques used to create a parallel simulation in Section 1.3, three categories are defined that group all of these techniques depending on the extent to which the parallelism is exploited: application-level parallelism, simulation-level parallelism and model-level parallelism. Those levels establish the level of knowledge of the specific system being modelled and simulated that is required to be applied by each technique. Our focus being on the simulation engine, it was determined that a simulation-level parallelism approach would best fulfil our objectives. A master-slave paradigm based on a centralised list with distributed events is adopted to create the framework for the parallel simulation, comprising a master *Event Manager (EM)* to handle the Future Event List (FEL) and the Virtual Clock (VC), and a set of slave *EEs*, in charge of executing the simulation events.

Explore techniques to reduce resource contention in a shared-memory parallel simulation

The way resource contention affects the parallel architecture designed is studied in Section 4.2 by focusing on two main situations:

1. several `Elements` simultaneously requesting different `Activities` that use the same `Resource` types;
2. and several `Elements` simultaneously requesting different `Activities` that use the same `Resource`.

The first problem involves the static structure of the model. *Activity managers* are introduced as a mechanism for creating a static partition of the model that allows protecting the concurrent access to shared simulation components only when needed.

The second problem stems from the rich definitions of the `Resources`' timetables, which allow the same `Resource` to be available to several `Resource` Types

simultaneously. Since timetables are dynamically interpreted as the simulation time advances, a dynamic solution is proposed that detects and later arbitrates the conflicts for Resources. This mechanism, built on top of the activity managers, is based on the construction of *Conflict zones* that dynamically detect the conflicts caused by several Elements trying to capture the same Resource. After the conflicts are detected, this structure arbitrates the access of the contenders to the Resources by means of a dynamically created stack of synchronisation mechanisms.

Analyse techniques and algorithms that improve simulation performance

Sections 4.6 to 4.13 are devoted to further enhancing and optimising the parallel framework proposed by applying different techniques and algorithms. The following enhancements are studied and their effect on overall performance analysed:

1. Execution of the EEs in a Java thread pool.
2. Integration of the thread pool into the simulator code.
3. Exploitation of event locality by means of local buffers both for the FEL and for events currently in execution.
4. Block dispatching, that is, grouping the events before they are sent to the EEs for execution.
5. Redefinition of events and partition of the execution phase by drawing inspiration from concepts in the 3-phase algorithm.
6. Development of a hybrid scheme Event Manager Executor (EME), where the EM also executes events.
7. Fine-grain adjustment of those implementation details that have a strong influence on execution time.

An analysis of the performance of the different enhancements shows that using an external pool is not capable of outperforming sequential `SIGHOS`, except in very favourable scenarios. The integration of the thread pool into the simulator code improves the results but the scalability remains very poor. The simulator starts to become scalable only after event locality is applied.

Though block dispatching reduces the execution time even more, a modified version of the Three-Phase approach algorithm described in Subsection 1.2.4 is introduced that exhibits the best behaviour for all the scenarios. This algorithm redesigns the events used in the simulation to 1) remove several synchronisation structures required in the original algorithm and 2) reduce the number of redundant actions performed by simulation clock tick.

A curious effect appears when a hybrid EME scheme is applied in order to fully utilise all the available processors: though the block dispatching version outputs worse results, the like-3-Phase algorithm profits from this scheme.

Finally, several implementation details that heavily influence the final performance are examined and compared in terms of their practicality.

- Using an atomic variable in a spinlock is proven to notably outperform a passive wait in a Java semaphore.
- Extending the `Thread` class is shown to be extremely inefficient when compared to implementing the `Runnable` interface.
- The usage of a specific barrier implementation notably affects the performance of the simulation. The poor efficiency of the standard Java implementation is also demonstrated.

Establish the potential of a generic PDES to model and simulate organisations

Achieving the previous goal makes possible a practical implementation of a generic PDES for BPS by using `SIGHOS` as a case study. The resulting parallel

version of SIGHOS (PSIGHOS) is used to establish the effects of the model characteristics on the performance of the proposed solution:

- “Saturated” systems, with a large rate of simultaneous events per simulation clock, benefit from this approach.
- PSIGHOS events are able to be customised so as to allow complex operations to be performed during the simulation. Our approach benefits from a higher computing load.
- Loosely coupled systems with specialised resources and activities create more activity managers, which increase the amount of parallelism exploitable during the simulation.

Establish the suitability of Java as a language for implementing the proposed simulation

Section 2.8 reviews the main characteristics that make Java a suitable language for DES, including object-orientation, portability, the lack of pointers, multi-threading support and its network-aware architecture. Further, all of Chapter 2 highlights the role of Java as a suitable language for dealing with the most important techniques utilised to implement the process-oriented worldview.

Study the performance of this kind of parallel simulator in a multi-core computer

PSIGHOS is also used to study the performance of a PDES tool based on the hybrid EME architecture, the like-3-phase algorithm, and to fine tune several implementation details. A parameterisable test benchmark is defined to be run on a 16-core test platform with 32 GB RAM.

In the best case scenarios, the speedup over the sequential simulation exceeds a value of 9 for 16 threads, even offering superlinear results for 4 threads. Clearly, the generic tool developed is potentially capable of profiting from the resources of a multi-core computer.

Validate the usefulness of the approach by applying it to a real-world problem

Chapter 5 presents a prototype simulation model of a whole hospital. Said prototype is intended to be used as a case study to corroborate both the modelling capabilities and performance of PSIGHOS when applied to a real-world problem. The chapter starts by describing the main components of a hospital as a system and the utility of modelling and simulating in aiding hospital management to measure the performance of their organisation. Patient *wait times* are established as one of the few objectively quantifiable Performance Indicators (PIs).

A specific hospital was chosen due to its long history of cooperation with the Simulation Group from the Universidad de La Laguna in this area. Hence, a nationally awarded simulation model of the Hospital Universitario Nuestra Señora de Candelaria (HUNSC)'s general and digestive surgical department is depicted as an earlier contribution of said research group.

A brief survey of the literature shows the difficulties of modelling a whole hospital by reviewing some earlier contributions. Special attention is devoted to Dr. Günal's thesis (Günal, 2008) and, more specifically, to HADA, a tool developed by both Dr. Günal and the author of this thesis. HADA analyses two different data sources (the UK nationally collected Health Episode Statistics (HES) and local Patient Administration System (PAS)) as a way of understanding a hospital's past performance as well as for estimating parameters for a hospital simulation model.

Having established a proper background for creating a hospital model, a patient-centred conceptual model of the whole hospital is presented. A *divide and conquer* approach is applied to simplify the modelling by creating four functional submodels: an Accident and Emergency department, Surgical Departments, Medical Departments and Central Services. This division allows the modeller to focus on the specific needs of each submodel. Since PSIGHOS' performance and modelling capabilities could be validated anyway, the Accident and Emergency (A&E) department as well as emergency patients were deleted from the final model for the sake of simplicity, which, while it prevented the model

from being fully validated for practical applications, considerably shortened the development time.

A highly parameterisable computational model is implemented with PSIGHOS so as to prove its modelling capabilities.

- Complex resource behaviours, such as multi-tasking, complex timetables and specialisation are captured.
- The combined usage of workflows and `Element` types is shown to be a suitable mechanism for modelling the sophisticated behaviour of the system's entities, that is, the patients of the hospital.
- Two types of activities are used: simple tasks whose length is defined based on a predefined time function; and compound tasks, which are considered complete once their associated subprocess finishes.

PSIGHOS' performance is checked by designing several scenarios for different problem sizes. Scalability does not increase with the number of processors since the *activities* involving central services, laboratories and the shared surgical resources (Postanaesthesia Care Unit (PACU), Intensive Care Unit (ICU) and anaesthetists) bottleneck the simulation. Considering that the model is generic and not especially adapted to exploit a parallel platform, the performance observed is relatively satisfactory for fewer than 4 threads.

Further Research

The parallel simulation tool developed over the course of this thesis shows excellent performance when applied to the different scenarios tested with the synthetic model introduced in Section 4.4. A satisfactory result is also obtained when dealing with a more realistic problem, like the one proposed in Chapter 5. Further research is required, however, if new approaches are to be found that are able to enhance performance by benefiting from the strengths of PSIGHOS, that is, *generality* and *automated exploitation of parallelism*.

Temporal Uncertainty

The main drawback of the parallel architecture developed in this thesis is its zero-lookahead nature. Hence, only events with timestamps that are strictly simultaneous can be parallelly executed. Fujimoto (1999) studies this problem in federated simulation systems, which are a widely used example of domain decomposition in PDES. He proposes a synchronisation algorithm for exploiting temporal uncertainty that is later refined by Loper (2002), and that reduces the number of global synchronisation computations required to causally order the events in a domain-decomposition parallel simulation. Considering that temporal uncertainty is inherent in the length of the activities as defined by PSIGHOS, this approach seems a suitable mechanism for increasing the number of events that may be concurrently processed. Nevertheless, three main problems arise that must be solved before temporal uncertainty can be applied to PSIGHOS.

1. Would it be possible to adapt this technique to a different parallel architecture? In other words, would simulation-level parallelism profit from this technique?
2. Would PSIGHOS' modelling interface remain unaltered? Something similar to the pre-sampling algorithm, introduced by Loper (2002), would be required to prevent a user from explicitly declaring time intervals while maintaining an uncertainty based on probability distributions.
3. The third problem derives from the like-3-phase algorithm used by PSIGHOS. Initially, since two synchronisations are involved per simulation clock tick, this algorithm precludes the use of approximate time. Hence, this technique would not profit from the advantages of the most efficient PSIGHOS algorithm.

Time Decomposition

A completely different technique is to apply time instead of domain decomposition (Subsection 1.3.6). A steady-state simulation would benefit from this approach insofar as the adjustment errors between successive time intervals could be ignored. Kiesling (2006) proposes an innovative technique that relies on combining time decomposition and a progressive refinement of results. Hence, a simulation can output imprecise results very quickly and, if more accurate results are required, the simulation can be allowed to continue running. Since Kiesling presents an application example with a queuing simulation, it is reasonable to consider its application to the simulation of organisations.

GPU-Based Simulation

Turning our attention to state-of-the-art technologies, the use of Graphics Processing Units (GPUs) is becoming popular among the simulation community. Park and Fishwick (2008) utilise an algorithm that is very similar to the pseudocode introduced in Listing 4.1. Their approach takes a list of events and sends it to the GPU processors. The list not only selects events with the same timestamp, but a modified relaxed synchronisation is used that allows all the events within a time interval to be concurrently executed at the expense of accuracy. Then, a detection and compensation technique adapted to the specific system being simulated is utilised to minimise the error. The case studies proposed by the authors of this paper are very simple, but combining their ideas with the approach presented in this thesis seems like a sensible idea.

Advances in Real Applications

Focusing on real applications, there are several fields in which PSIGHOS can be enhanced.

- PSIGHOS lacks of a graphical interface. A Master's thesis currently being written by Rayco Díaz Batista is expanding on the work developed in

(Muñoz, 2006) in order to obtain a flexible and easy-to-use interface where a user is able to define the simulation components and launch basic simulation experiments.

- Though simulation itself is a useful tool for organisation management, it is highly desirable to provide said management with tools that help them in their decision-making processes. Hence, integrating simulations in control panels or holistic business solutions offers management the best opportunities for achieving their business goals. Earlier contributions by our Simulation Group, such as (Aguilar et al., 2005a), rely on the combined use of a Multi-Agent System (MAS) and a simulation as a tool to aid in the decision-making processes. Such an approach presents a promising line of research.
- Although PSIGHOS is not currently intended to be used in a distributed environment, future releases of the tool will take advantage of the net. Actually, there is a Master's thesis (García-Hevia, 2008) that describes how to interact with the simulator by using web services.
- In this thesis, PSIGHOS has been applied to model a hospital. However, several other systems can be modelled and simulated with this tool. Several examples actually exist that show some real applications of the simulator, such as helpdesks and call centres (Baquero et al., 2005; Castilla et al., 2007) or eGovernment process development (Callero and Aguilar, 2009).

Resumen

En este anexo se presenta un resumen en español de la tesis con especial énfasis en las contribuciones y conclusiones de este trabajo.

A.1. Planteamiento del Problema

Las organizaciones empresariales han ido asimilando con el paso de los años que la información es uno de sus bienes más preciados. Las Tecnologías de la Información (TI) se han erigido como una importante fuente de inversión tanto para las empresas privadas como para las públicas. El resultado es una búsqueda de la mejora de los mecanismos que les permiten manejar la información y obtener rédito de ella, ya sea en términos económicos o de calidad.

Una visión global de la organización requiere de modelos que permitan resaltar los aspectos fundamentales que influyen en el correcto funcionamiento de la empresa, y descartar aquéllos que no aporten información valiosa al proceso de su comprensión y análisis. Los procesos de negocio, tal como los define (WfMC, 1999), aparecen como una visión de alto nivel centrada en el mercado que se prestan con facilidad a los objetivos deseados.

La simulación de estos procesos de negocio (SPN) es una herramienta que se emplea dentro del campo de la reingeniería de procesos de negocio (RPN) (Mut-hu et al., 2006) con dos objetivos fundamentales (Wynn et al., 2008): 1) analizar

el comportamiento de un proceso mediante el desarrollo de modelos de simulación precisos; y 2) comprender lo que ocurre al instaurar estos procesos en la organización mediante la ejecución de experimentos de simulación.

Una herramienta de SPN requiere:

- Bloques básicos para la construcción de un modelo: actividades, entidades, recursos y conectores.
- Estructuras que permitan ordenar la ejecución de las actividades: separaciones, uniones, ramificaciones y sincronizaciones.
- Funciones avanzadas de modelado, como atributos, expresiones y planificaciones horarias para los recursos.

Tal como demuestra (Bosilj-Vuksic et al., 2007), la simulación de eventos discretos (SED) se adapta perfectamente a los requerimientos listados.

Desafortunadamente, los modelos de simulación de los procesos de negocio son generalmente complejos debido al tamaño de las organizaciones y la existencia de múltiples relaciones entre las diferentes entidades de la empresa. Esta complejidad desemboca habitualmente en simulaciones muy “pesadas” que requieren tiempos de ejecución elevados. La simulación paralela incluye un conjunto de técnicas que permiten reducir el tiempo de ejecución de una simulación al aprovechar múltiples procesadores conectados por algún tipo de red de comunicaciones. Generalmente, una simulación paralela se ejecuta sobre procesadores físicamente cercanos que comparten memoria y dispositivos de entrada/salida. Estas plataformas se caracterizan por una latencia de comunicación, definida como el tiempo que tarda un mensaje en llegar de un procesador a otro) relativamente baja.

Las plataformas que quieren explotarse en esta tesis son los sistemas multi-núcleo. Un sistema multi-núcleo consiste en dos o más núcleos de procesamiento ubicados en el mismo “paquete” o circuito integrado. Los diferentes núcleos comparten ciertas estructuras, como las cachés de segundo o tercer nivel y buses

de entrada/salida. Habiendo mencionado la idoneidad de la SED para la simulación de procesos de negocio, la elección lógica para aplicar paralelismo es la simulación paralela de eventos discretos (SPED).

Si se atiende a la literatura existente, hasta ahora la SPED se ha restringido al dominio científico o militar y, casi siempre, se ha aplicado a modelos de sistemas muy concretos (Perumalla, 2006), lo que impide la reutilización de proyectos terminados para enfrentarse a nuevos problemas. La disponibilidad de una plataforma económica y accesible donde poder ejecutar aplicaciones paralelas, tal como es un ordenador multi-núcleo, presenta una oportunidad única de investigar en el aprovechamiento de estas tecnologías para mejorar la experiencia de los usuarios con las aplicaciones de simulación. Teniendo en cuenta, además, que la utilización del paralelismo requiere una amplia experiencia en lenguajes y paradigmas de programación que no son triviales, se plantea en esta tesis la búsqueda de mecanismos que permitan una explotación del paralelismo transparente al usuario.

Sólo dos trabajos previos tratan de avanzar en la mejora del rendimiento de la simulación de procesos de negocio a través del uso del paralelismo. Ferscha y Richter (1996) utiliza redes de Petri para modelar los procesos de negocio, que son posteriormente simulados con un motor de simulación masivamente paralelo. Zarei (2001) propone también el uso de la SPED para reducir el tiempo de ejecución de una SPN modelada usando grafos de flujo de control. Ambos trabajos presentan un pequeño caso de estudio y aplican descomposición por dominio (Fujimoto, 2000) como la técnica para explotar el paralelismo.

Desde el punto de vista del autor de esta tesis, estos trabajos son mejorables en dos aspectos fundamentales. Por un lado, el uso de formalismos o constructores demasiado abstractos (como son las redes de Petri o los grafos de flujo de control) implica que una experiencia notable en el campo del modelado de procesos de negocio no es suficiente para poder trabajar con estas herramientas. Sería deseable disponer de herramientas de modelado que usaran un lenguaje más cercano al del usuario final sin por ello comprometer el poder expresivo o la eficiencia del simulador. Por otro lado, ambas alternativas requieren de un estudio

del sistema para exponer explícitamente el paralelismo durante la creación del modelo conceptual. Esto limita la portabilidad y reutilización de las soluciones.

A.2. Objetivos de la Tesis

Teniendo claro el contexto de este trabajo de investigación, se han establecido los siguientes objetivos:

1. Identificar las características que hacen de la simulación de eventos discretos y, más específicamente, la orientación al proceso, una aproximación adecuada al modelado y simulación de procesos de negocio.
2. Discutir la viabilidad de la explotación automatizada del paralelismo en modelos de procesos de negocio.
3. Explorar técnicas que reduzcan la incidencia de los problemas de contienda por recursos en una simulación paralela en una plataforma de memoria compartida, tal como son los sistemas multi-núcleo.
4. Establecer el potencial de una simulación paralela de eventos discretos genérica para simular organizaciones.
5. Analizar las técnicas y algoritmos que mejoran el rendimiento de la simulación.
6. Establecer la adecuación de Java como lenguaje para implementar la simulación propuesta.
7. Estudiar el rendimiento de este tipo de simulación paralela en ordenadores multi-núcleo.
8. Validar la utilidad de esta aproximación mediante su aplicación a un problema real basado en la simulación de un centro hospitalario.

A.3. Resultados y Contribuciones

Atendiendo a los objetivos planteados, las contribuciones de esta tesis pueden resumirse en los siguientes apartados:

Identificar las características que hacen de la simulación de eventos discretos y, más específicamente, la orientación al proceso, una aproximación adecuada al modelado y simulación de procesos de negocio.

Después de estudiar las distintas aproximaciones a la SED, se ha comprobado que la orientación al proceso (o interacción de procesos) es la orientación más adecuada para modelar y simular los procesos de negocio. Por un lado, esta orientación ofrece constructores de alto nivel que permiten al usuario abstraerse de los detalles de implementación. Por otro lado, la interacción de procesos utiliza estructuras que permiten representar muy fácilmente los componentes de un modelo de procesos de negocio. Para verificar estas conclusiones se ha construido SIGHOS (Muñoz y Castilla, 2010), una librería para la simulación de procesos de negocio que utiliza los conceptos relacionados con los patrones de flujos de trabajo (Van Der Aalst et al., 2003) y (WfMC, 1999) como referencia para definir constructores que permiten la creación de modelos usando terminología más próxima al experto en la gestión de organizaciones SIGHOS define, entre otros, los siguientes componentes:

- **Recurso.** Cualquier bien material o humano requerido para realizar una tarea. Los recursos emplean con una serie de “entradas horarias” para establecer no sólo sus horas de disponibilidad, sino los “roles” para los que estarán disponibles. Los médicos o las máquinas de rayos X de un hospital son ejemplos de recursos.
- **Tipo de recurso.** Es cada rol o categoría a la que se asocia un recurso. Las tareas se definen en función a estos roles y no a los recursos específicos que pueden realizarlas. En el esquema que se propone, un mismo recurso puede tener varios roles simultáneamente, estando, por tanto, potencialmente

disponible para realizar varias tareas diferentes. Un cirujano puede tener horas de cirugía y también horas de consulta de su especialidad; podrían definirse, por tanto, un rol “cirujano” y un rol “doctor en consulta”.

- Grupo de trabajo. Describe un conjunto de pares <tipo de recurso, cantidad> que son necesarios para realizar una tarea. Un equipo quirúrgico se compone de uno o más cirujanos, una enfermera de instrumentación, el anestesista... Todos ellos conformarían un equipo de trabajo.
- Actividad. Describen cualquier tarea que requiere un tiempo determinado y uno o varios recursos para ser terminada. Los recursos que requiere una actividad se definen a través de los grupos de trabajo, pudiendo la misma actividad ser llevada a cabo con diferentes grupos y diferentes duraciones. Una intervención quirúrgica, una consulta con un médico, la atención de una llamada a un centro de atención al usuario (CAU)... pueden ser modelados como actividades.
- Flujo de trabajo. Una descripción del orden en que deben ejecutarse las actividades del sistema para llegar a un fin determinado. Por ejemplo, el flujo típico de un paciente que va a ser operado se podría componer de una actividad de preanestesia, una serie de actividades preoperatorias, la intervención quirúrgica en sí, la recuperación en la URPA y la posterior estancia en planta hasta recibir el alta.
- Entidad. Las entidades (o elementos) son instancias de los procesos definidos mediante los flujos de trabajo. Así, el ciclo de vida de una entidad consiste en la finalización de su flujo de trabajo asociado. Teniendo en cuenta el ejemplo de flujo de trabajo, un paciente sería tratado como entidad de la simulación, aunque podría aplicarse a conceptos más abstractos como un documento de un sistema de gestión documental o una incidencia en un CAU.

- Generador de elementos. Las entidades aparecen en el sistema bien obedeciendo un patrón temporal (se puede suponer que llega un paciente cada 3 minutos de media al servicio de urgencias), bien en respuesta a algo que ocurre en el sistema (la recepción de muchas incidencias del tipo “fallo al leer mi correo electrónico” pueden llevar a la generación de una incidencia “chequear servidor de correo de la organización”).

Discutir la viabilidad de la explotación automatizada del paralelismo en modelos de procesos de negocio.

Se han establecido dos hitos que, desde el punto de vista del autor de esta tesis, permiten la explotación automatizada del paralelismo: 1) el aislamiento entre la definición del modelo y la ejecución de la simulación; y 2) el uso de una aproximación al paralelismo enfocada en el motor de simulación y no en el modelo específico que va a simularse.

Para conseguir el primer hito, se ha planteado el uso de una traducción automática de procesos a eventos para la implementación a bajo nivel de la simulación. Esta transformación permite mantener una interfaz de modelado de alto nivel con un lenguaje más cercano al que define el problema (procesos de negocio), mientras deja el bajo nivel en manos de un motor de simulación más eficiente y flexible basado en eventos.

Con respecto al segundo hito, se han establecido tres categorías para agrupar las diferentes técnicas empleadas para paralelizar una simulación, dependiendo del nivel de dependencia de la técnica con las características del sistema bajo estudio. De esta manera, se puede hablar de paralelismo a nivel de aplicación, de simulación o de modelado. El nivel de aplicación no utiliza (o emplea muy poco) conocimiento específico sobre el motor de simulación o el modelo, y permite claramente el desarrollo de soluciones que no requieran la intervención del usuario en este proceso. Sin embargo, al ser soluciones demasiado genéricas, la ganancia potencial de esta aproximación es bastante limitada. En el otro lado del espectro, el nivel de modelado permite obtener las mejores ganancias a costa de un diseño explícito del modelo conceptual primero, y computacional después,

que exponga el paralelismo presente en el sistema bajo estudio. El nivel de simulación modifica el motor de simulación sin tener en cuenta las características del modelo específico a simular y permite la construcción de soluciones que puedan enfrentarse a cualquier problema de forma genérica. El equilibrio entre generalidad y eficiencia que ofrece esta aproximación es el principal motivo que ha llevado al autor de esta tesis a optar por estas técnicas.

Para implementar esta aproximación se ha planteado inicialmente un esquema basado en el paradigma maestro-esclavo, en el que un gestor de eventos “maestro” maneja la lista de eventos futuros y el reloj de simulación, mientras un conjunto de hilos “esclavos” ejecutan los eventos.

Explorar técnicas que reduzcan la incidencia de los problemas de contienda por recursos en una simulación paralela en una plataforma de memoria compartida, tal como son los sistemas multi-núcleo.

Existen dos problemas fundamentales con respecto al acceso a los recursos de simulación compartidos en el esquema paralelo que se plantea:

1. Varias entidades pueden estar solicitando simultáneamente diferentes actividades que usan, en alguno de sus grupos de trabajo, el mismo tipo de recurso.
2. Varias entidades pueden estar solicitando simultáneamente diferentes actividades que tienen disponible, para alguno de los tipos de recurso de alguno de sus grupos de trabajo, el mismo recurso.

El primer problema tiene que ver con la estructura *estática* del modelo. Para solucionarlo se ha planteado el uso de los *Gestores de actividades*, que son un mecanismo para crear una partición estática del modelo. Cada gestor de actividades agrupa las actividades que usan los mismos tipos de recurso y añade estructuras para proteger el acceso concurrente a estos componentes de simulación.

El segundo problema se deriva de la forma de definir los horarios de los recursos que, tal como se han planteado, permiten una gran flexibilidad que incluye

la posibilidad de que un recurso esté disponible simultáneamente para varios tipos de recurso o roles. Como los horarios son interpretados dinámicamente por el motor de simulación, es necesaria una solución que detecte primero y arbitre después cualquier posible conflicto también de forma dinámica. Este mecanismo, que usa como base los gestores de actividades, se basa en la identificación de los conflictos causados por múltiples entidades tratando de acceder al mismo recurso. Una vez detectado el conflicto, se construye una *zona de conflicto* que sirve para ordenar el acceso de las diferentes entidades al recurso mediante una pila de mecanismos de sincronización.

Analizar las técnicas y algoritmos que mejoran el rendimiento de la simulación.

Una parte considerable de esta tesis se ha dedicado a proponer técnicas y algoritmos que mejoraran el rendimiento del esquema paralelo propuesto. Las siguientes mejoras han sido propuestas y su impacto en el rendimiento de la simulación estudiado:

1. Explotación de la localidad de los eventos mediante el uso de cachés locales tanto para los eventos futuros como para los eventos actualmente en ejecución.
2. Envío en bloque de los eventos, es decir, los eventos se agrupan antes de ser enviados a los ejecutores de eventos.
3. Redefinición de los eventos y partición de la fase de ejecución tomando como inspiración conceptos de la aproximación en tres fases (Pidd, 1998).
4. Desarrollo de un esquema híbrido en el que el gestor de eventos también ejecuta eventos.
5. Ajuste fino de los detalles de implementación que tienen una mayor influencia en el rendimiento final.

El análisis de las diferentes mejoras mostró que la explotación de la localidad de los eventos ofrecía unos resultados que mejoraban los tiempos de ejecución

del equivalente secuencial, a la vez que ofrecía una moderada escalabilidad a medida que se aumentaba el número de hilos de ejecución.

La técnica del envío de eventos en bloque permitió reducir un poco más el tiempo de ejecución, pero fue la redefinición del algoritmo de simulación que adoptaba los conceptos de la simulación en tres fases la que supuso un salto cualitativo notable. Este nuevo algoritmo rediseñaba los eventos de la simulación eliminando algunas de las estructuras de sincronización requeridas por el algoritmo original y reduciendo el número de acciones redundantes ejecutadas en cada ciclo de reloj.

El esquema híbrido no aportó ninguna mejora al algoritmo original pese a aprovechar un hilo más para ejecutar eventos. Sin embargo, el algoritmo similar a las tres fases sí pudo aprovecharse de este nuevo esquema.

Entre los detalles de implementación estudiados, cabe destacar:

- El uso de una variable atómica en una espera activa mostró un rendimiento superior al de las esperas pasivas implementadas con semáforos de Java.
- Se demostró que crear objetos para ejecutar tareas que extendieran la clase Thread era considerablemente más ineficiente que implementar la interfaz Runnable.
- La implementación específica de las barreras de sincronización usadas en la simulación afectaba notablemente al tiempo de ejecución. Se pudo comprobar que la implementación estándar incluida en Java ofrecía una escalabilidad muy pobre.

Establecer el potencial de una simulación paralela de eventos discretos genérica para simular organizaciones.

Mediante la consecución de los objetivos anteriores fue posible la implementación práctica de PSIGHOS, un SPED genérico para la simulación de procesos de negocio basado en SIGHOS. Al explotar automáticamente el paralelismo, se

estudió la influencia de las características del modelo en la eficiencia de la simulación:

- Los sistemas “saturados”, con una gran tasa de eventos simultáneos se benefician de este tipo de simulación paralela.
- Los eventos de PSIGHOS disponen de “anclajes” donde añadir código de usuario para realizar cálculos o modificar atributos de la simulación. Incrementar las exigencias de procesamiento es beneficioso para esta aproximación.
- Los sistemas débilmente acoplados, en los que hay más recursos y actividades especializados aumentan el número de gestores de actividades. Esto, a su vez, incrementa el paralelismo potencial que puede ser explotado durante la simulación.

Establecer la adecuación de Java como lenguaje para implementar la simulación propuesta.

En numerosos trabajos se ha justificado la idoneidad de Java como lenguaje de programación para implementar herramientas de SED (Cassel y Pidd, 2001; Ferscha y Richter, 1997; Jacobs et al., 2002; Martin, 1997). En general, se destaca el uso que hace de la orientación a objetos, su portabilidad y la ausencia de punteros. El que sea un lenguaje construido con primitivas para manejar hilos, lo convierte en una más que razonable alternativa para enfrentarse a una aplicación que va a ejecutarse en una máquina con múltiples núcleos.

Estudiar el rendimiento de este tipo de simulación paralela en ordenadores multi-núcleo.

Para medir el rendimiento de PSIGHOS se diseñó un problema sintético que tratara de poner de relevancia las diferentes características que podían tener un mayor efecto en el tiempo de ejecución de la simulación. De esta manera, se definieron parámetros para establecer el ratio de eventos simultáneos, la escala del

problema, el número de gestores de actividades que se iban a crear, la proporción de recursos en situación de crear conflictos, etc. Los diferentes escenarios se pusieron a prueba en una máquina con 16 núcleos y 32 GB de RAM.

La figura A.1 muestra la aceleración obtenida sobre la ejecución del mismo escenario con SIGHOS para distintas configuraciones de parámetros. El eje x representa el número de ejecutores de eventos utilizados en cada prueba.

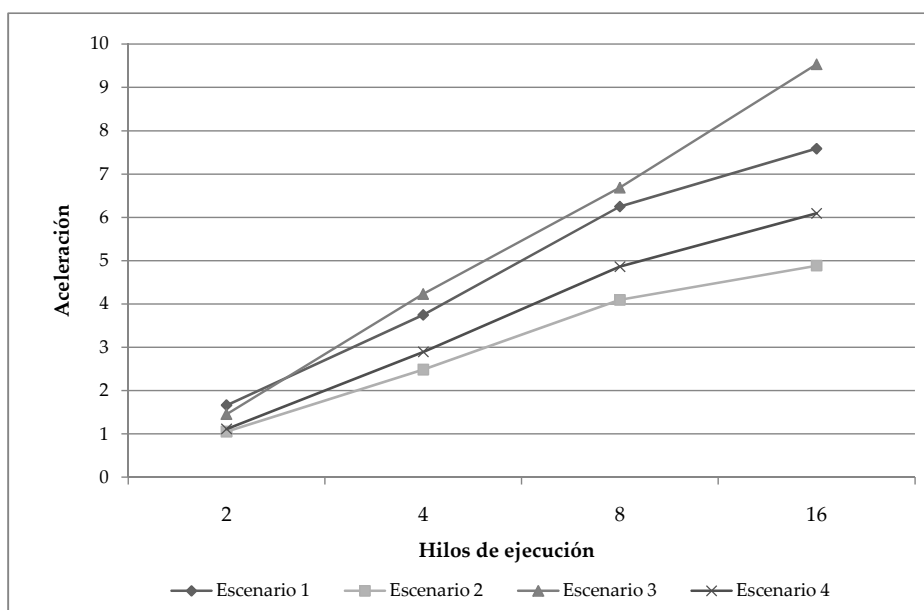


Figura A.1. Aceleración de PSIGHOS sobre SIGHOS

Validar la utilidad de esta aproximación mediante su aplicación a un problema real basado en la simulación de un centro hospitalario.

Para comprobar en un escenario más realista la eficiencia y capacidad expresiva del simulador, se ha presentado como caso de estudio un modelo del flujo de pacientes a través de un hospital. El modelo recoge aportaciones de (Aguilar, 1998) y (Günel, 2008), pero introduce una nueva perspectiva en la que se realiza

una descomposición funcional de la organización que remarca la especialización de los recursos del hospital. De esta manera, el modelo, validado con personal del Hospital Universitario de Canarias (HUC), descompone el hospital en los siguientes módulos:

- Servicio de urgencias, que recibe los pacientes urgentes. Estos pacientes son dados de alta o ingresados en algún otro servicio del hospital.
- Servicios quirúrgicos, que comprende todos los servicios del hospital que realizan actividades quirúrgicas. La URPA y la UCI, compartidas por todos los servicios quirúrgicos, se incluyen también en este módulo.
- Servicios médicos, que incorpora todos aquellos servicios que no realizan intervenciones quirúrgicas. Los pacientes acuden a estos departamentos a recibir un diagnóstico o tratamiento médico.
- Servicios centrales, que proporcionan al resto de servicios los medios humanos y materiales para realizar todo tipo de pruebas clínicas tales como análisis, rayos X y medicina nuclear.

La figura A.2 esquematiza las relaciones entre los módulos. Se trata de un modelo genérico y altamente parametrizable que puede adaptarse a las características de diferentes centros.

Para el estudio de la eficiencia de los algoritmos desarrollados, se ha presentado un modelo computacional que omite el servicio de urgencias, así como la incidencia de los pacientes urgentes en general. Esto impide que el modelo computacional pueda utilizarse para obtener resultados reales, pero no obstaculiza su utilización para los fines previstos de medir la eficiencia de la simulación paralela y comprobar la capacidad expresiva de los objetos de modelado propuestos.

Con respecto a la capacidad expresiva, se ha comprobado que los objetos de modelado propuestos permiten capturar comportamientos complejos de los recursos humanos, como dedicación multi-tarea, horarios complicados y diferentes grados de especialización. Las entidades del sistema, es decir, los pacientes,

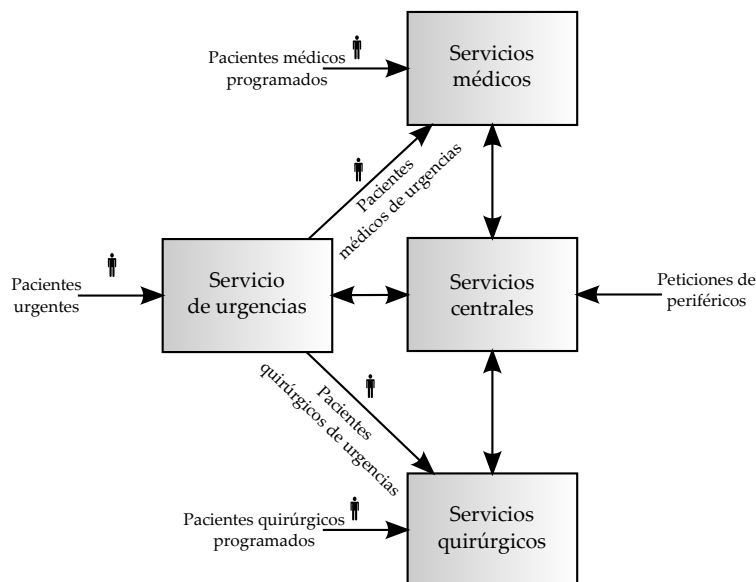


Figura A.2. Modelo conceptual del hospital

pueden ser agrupados en categorías de forma sencilla, permitiendo la distinción por patologías, gravedad del estado, etc. Además, cada tipo de paciente puede seguir un flujo de trabajo diferente. En el modelo también se usaron capacidades para definir tareas dependientes del tiempo y tareas cuya finalización depende de la terminación de un flujo de trabajo asociado.

Se probaron distintas escalas de modelo variando el número de servicios. Para ello, se definieron una serie de seis plantillas de servicios (servicio “similar” a dermatología, servicio “similar” a reumatología...) y se dimensionaron los escenarios en función de la cantidad de servicios que se creaban por plantilla. De esta manera, se probó con 1, 2 y 4 servicios por plantilla, es decir, 6, 12 y 24 servicios en toda la simulación. La configuración de los servicios centrales se mantuvo fija y similar a la información del equipamiento real del HUC.

Al tratarse de un sistema en el que las actividades de los servicios centrales hacen de cuello de botella, se pudo comprobar que, para un tamaño de proble-

ma moderado, la aceleración de la simulación escalaba bastante bien hasta los 4 hilos, tal como se ve en la figura A.3. Los tiempos seguían mejorando usando más hilos, pero de forma mucho más moderada.

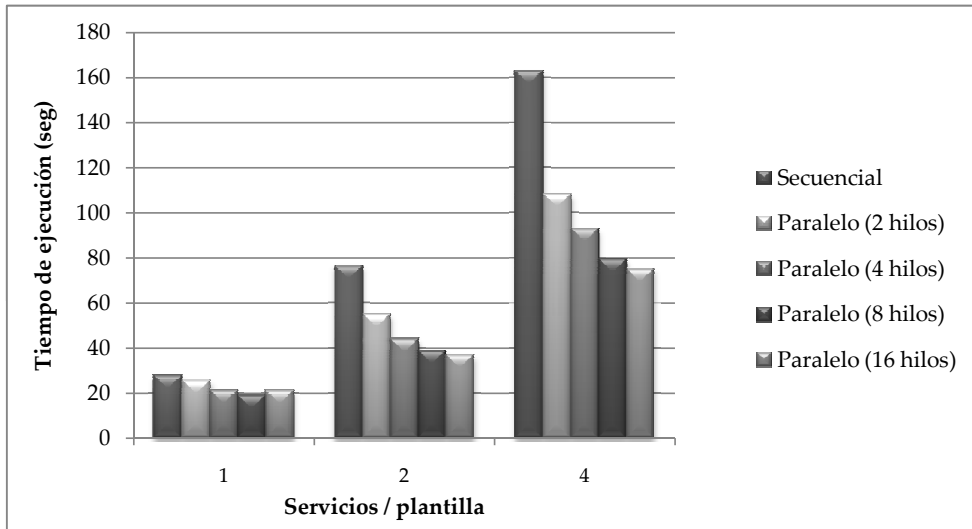


Figura A.3. Tiempo de ejecución para diferentes escenarios de un modelo completo de un hospital

A.4. Conclusiones

El trabajo realizado para esta tesis puede resumirse como la exploración de las posibilidades que ofrecen las arquitecturas multi-núcleo para la explotación automatizada del paralelismo en la simulación de modelos genéricos de procesos de negocio.

Tres conceptos destacan en el párrafo anterior:

1. *Generalidad*: Las soluciones planteadas se encuadran en el campo de la simulación de organizaciones pero no en la simulación de un modelo o problema específico. En su lugar, se han presentado los patrones de flujo de

trabajo como una alternativa viable para describir genéricamente los procesos de negocio de una organización. Esta tesis ha evidenciado que el uso combinado de los patrones de flujo de trabajo junto a la definición de estructuras de modelado de alto nivel adaptadas al lenguaje empresarial (con términos como actividades, recursos humanos u horarios) permite a un usuario con poca experiencia en el campo de la simulación familiarizarse rápidamente con los componentes requeridos para crear un modelo. Para corroborarlo, se ha desarrollado una implementación práctica de estos conceptos en una librería de simulación en Java llamada SIGHOS.

2. *Automatización:* A partir de una plataforma genérica para SPN, se ha explorado la explotación automatizada del paralelismo presente en los modelos de simulación en un ordenador multi-núcleo. En este caso, el concepto de automatización implica la ocultación al usuario final de los mecanismos que explotan el paralelismo en la simulación. Esta tesis ha presentado un nuevo algoritmo que distribuye automáticamente los eventos de simulación entre un conjunto de hilos ejecutores de eventos, poniendo especial atención a los problemas derivados de la ejecución concurrente de eventos que acceden a recursos compartidos de la simulación. El citado algoritmo combina conceptos de la aproximación en tres fases, el paradigma maestro-esclavo, y una arquitectura paralela que utiliza una lista de eventos futuros centralizada que posteriormente son distribuidos. Como en el caso anterior, una implementación práctica de este algoritmo llamada PSIGHOS ha sido presentada. PSIGHOS preserva la interfaz de modelado de SIGHOS, pero cambia el núcleo secuencial de la librería por un núcleo paralelo.
3. *Eficiencia:* La simulación de procesos de negocio es un problema en el que la contienda por recursos y la frecuente comunicación entre entidades limita normalmente el grado de explotación potencial del paralelismo. Esta tesis ha analizado estos problemas y presentado nuevos avances en la gestión estática y dinámica de los recursos compartidos para reducir la sobrecarga que produce la contienda por recursos en un entorno paralelo.

Los resultados experimentales obtenidos han corroborado la eficiencia de esta proposición al enfrentarse con problemas sintéticos, y han mostrado aceleraciones esperanzadoras frente a problemas genéricos y más realistas, como el caso de estudio presentado.

Estos tres conceptos se combinan de una manera única que no encuentra ejemplos similares en la literatura relacionada con el campo de la simulación de procesos de negocio.

A.5. Líneas Abiertas

Este trabajo ha demostrado que es posible obtener un excelente rendimiento con una herramienta genérica que explote automáticamente el paralelismo al enfrentarse a diferentes escenarios creados sintéticamente. No sólo eso, los resultados obtenidos cuando el escenario está planteado desde las premisas de un problema con las características de un sistema real son generalmente satisfactorios. Sin embargo, la búsqueda de una mayor eficiencia en este último caso, a la vez que se preservan los principios de generalidad y automatización, permiten plantear algunas líneas abiertas para esta investigación.

A.5.1. Incertidumbre Temporal

La mayor dificultad que tiene el algoritmo propuesto para explotar eficientemente el paralelismo en la simulación es su fuerte dependencia con el ratio de eventos simultáneos. Esto es así porque no puede utilizar, como hacen otras alternativas clásicas de la literatura, una predicción (lookahead) para poder ejecutar eventos cuya etiqueta de tiempo es mayor que el tiempo de simulación actual. Fujimoto (1999) propuso un algoritmo de sincronización para ser aplicado en sistemas federados de simulación, un ejemplo de la descomposición de dominio bastante usado en el ámbito militar. Dicho algoritmo, posteriormente refinado por Loper (2002) explotaba la incertidumbre temporal, es decir, la falta

de certeza del instante exacto de ocurrencia de un evento, para reducir el número de sincronizaciones necesarias para mantener el orden causal de los eventos. Dado que esta incertidumbre temporal puede encontrarse asociada a la duración de las actividades tal como las define PSIGHOS, este algoritmo podría ser aplicado para incrementar el número de eventos a procesar concurrentemente.

A.5.2. Descomposición Temporal

Existen dos ramas principales en la descomposición por dominio: descomposición espacial, que es la más frecuentemente usada y a la que dedica casi la totalidad de su libro Fujimoto (2000), y la descomposición temporal. Este segundo tipo de descomposición o división del modelo, simula en paralelo diferentes periodos de tiempo del sistema modelado, lo que puede ser útil en simulaciones estacionarias. Ciertamente, el mayor desafío de esta técnica es tratar adecuadamente el error que se produce en los puntos de unión entre periodos sucesivos. Kiesling (2006) propone una nueva técnica que combina la descomposición temporal y un refinamiento progresivo de los resultados. De esta manera, una simulación podría devolver un resultado impreciso muy rápidamente y, si se requieren resultados más exactos, continuar su ejecución el tiempo que sea preciso. El ejemplo que presenta Kiesling está basado en la simulación de colas, con lo que resulta una idea de posible aplicación en los problemas que trata esta tesis.

A.5.3. Simulación en Unidades de Procesamiento Gráfico

Si se atiende al estado de las investigaciones, las unidades de procesamiento gráfico (UPG o GPU en sus siglas inglesas) están ganando cada vez más adeptos entre los miembros de la comunidad científica dedicada a la simulación. Park y Fishwick (2008) proponen un algoritmo muy parecido al algoritmo maestro-esclavo que se presenta en esta tesis antes de aplicarle las sucesivas mejoras. Su algoritmo toma una lista de eventos y la distribuye entre un conjunto de procesadores de una UPG. En lugar de enviar únicamente eventos con la misma etiqueta de tiempo, el uso de un algoritmo de sincronización relajada permite que todos

los eventos en un intervalo de tiempo puedan ser ejecutados concurrentemente a costa de una ligera pérdida de precisión. Esto obliga a los autores a añadir un mecanismo de detección y compensación adaptado a cada problema específico que minimice el error. Aunque los casos de estudio que presenta ese artículo son bastante simples, parece bastante razonable pensar en combinar estas ideas con la propuesta de esta tesis.

A.5.4. Avances en Aplicaciones Reales

Atendiendo a las aplicaciones prácticas que pueden derivarse de esta tesis, hay varios campos en los que se puede mejorar PSIGHOS:

- PSIGHOS carece de interfaz gráfica. Actualmente, un proyecto de fin de carrera con el alumno de Ingeniería Informática Rayco Díaz Batista está expandiendo el trabajo de (Muñoz, 2006) para dotar al simulador de una interfaz fácil de usar y flexible con la que un usuario pueda acceder a todos los componentes de modelado y lanzar experimentos de simulación.
- Aunque la simulación puede usarse por sí misma como mecanismo para ayudar en la gestión de organizaciones, es altamente deseable disponer de herramientas que ayuden a los gestores en los procesos de toma de decisiones de forma más efectiva. Por tanto, la integración de la simulación en paneles de control o soluciones sofisticadas de gestión de procesos de negocio ofrece a los gestores una mejor oportunidad de conseguir sus objetivos empresariales. Trabajos anteriores del Grupo de Simulación de la ULL, tales como (Aguilar et al., 2005a) ya han presentado el uso combinado de los sistemas multi-agente con la simulación como herramienta de ayuda a la toma de decisiones, y constituyen un camino a seguir.
- Aunque PSIGHOS no está actualmente enfocado a su uso en entornos distribuidos, trabajos como el proyecto de fin de carrera de Yurena García-Hevia Mendizábal (García-Hevia, 2008) describen cómo interactuar con el simulador usando servicios web.

- Aunque el caso de estudio presentado en esta tesis está enfocado a la gestión hospitalaria, PSIGHOS ha sido utilizado con éxito en estudios de simulación enfocados a otros campos como centros de atención al usuario (Baquero et al., 2005; Castilla et al., 2007) o estudio de los procedimientos electrónicos en la administración pública (Callero y Aguilar, 2009).

Appendix

B

PSIGHOS: User Interface

PSIGHOS includes a comprehensive set of modelling components. This appendix reviews those modelling classes included in PSIGHOS, and enumerates the most relevant attributes and methods, the “hooks” added so that the user is able to incorporate customised behaviour, and the pieces of information that the object produces and the simulation engine collects.¹

Before describing the modelling classes, let us review the package structure of PSIGHOS:

- *es.ull.isaatc.simulation* Main modelling classes and the core classes for defining time units, timestamps, etc.
- *es.ull.isaatc.simulation.flow* Classes required to define workflows.
- *es.ull.isaatc.simulation.condition* The Condition interface and some predefined conditions.
- *es.ull.isaatc.simulation.variable* The Variable interface and some primitive type variables.
- *es.ull.isaatc.simulation.info* Classes declaring the pieces of information that the simulation emits and a user can handle by means of the InfoReceiver interface.

¹Since it describes the modelling components, this API can be used for SIGHOS as well.

- *es.ull.isaatc.simulation.inforeceiver* Core classes and interfaces that handle the information produced by the execution of the simulation model.
- *es.ull.isaatc.simulation.inforeceiver.view* Some predefined views that facilitate the processing of information produced by the execution of the simulation model.
- *es.ull.isaatc.simulation.factory* Factories of simulation objects, required when dynamic compilation is needed, or in case the simulation is going to be created from other application.
- *es.ull.isaatc.simulation.test.** Test classes.
- *es.ull.isaatc.function* The core classes used to define temporal behaviours.
- *es.ull.isaatc.util* Several utilities, such as cycle classes and statistical functions.

Table B.1. Class TimeUnit

Description
Different time units that can be used in a simulation: <code>MILLISECOND</code> , <code>SECOND</code> , <code>MINUTE</code> , <code>HOUR</code> , <code>DAY</code> , <code>WEEK</code> , <code>MONTH</code> and <code>YEAR</code> .

Table B.2. Class Timestamp

Description
A time value expressed as a pair <code><TimeUnit, long></code> .
Attributes
<code>TimeUnit unit</code> The time unit of the value
<code>long value</code> A time value expressed in the corresponding unit
Methods
<code>Timestamp getWeek()</code> Returns a "1 WEEK" timestamp
<code>Timestamp getDay()</code> Returns a "1 DAY" timestamp
<code>Timestamp getHour()</code> Returns a "1 HOUR" timestamp
<code>Timestamp getMinute()</code> Returns a "1 MINUTE" timestamp
<code>Timestamp getZero()</code> Returns a "0" timestamp

Table B.3. Class TimeFunction**Description**

A wrapper class that returns values corresponding to a specified function, which can depend on time and follow a random number distribution.

Table B.4. Class PeriodicCycle**Description**

A periodically repeated sequence of events. It can be defined in two different ways: on the one hand, a cycle can be defined that stops when a particular timestamp (endTs) is reached; on the other hand, a cycle can be defined that stops after a specific number of iterations. One may even define an infinite cycle (0 iterations). Cycles can contain subcycles, which are launched within the period of the parent cycle.

Attributes

TimeStamp startTs	Relative time when this cycle is expected to start
TimeFunction period	Time interval between two successive occurrences of an event
TimeStamp endTs	Relative time when this cycle is expected to finish
int iterations	How many times this cycle is executed. 0 indicates infinite iterations
Cycle subCycle	Subcycle started within every period

Table B.5. Class TableCycle

Description	
A set of instants when something happens. A subcycle can be defined that is launched between every two consecutive activations of the parent cycle.	
Attributes	
TimeStamp[] timestamps	The activation instants for this cycle
Cycle subCycle	Subcycle started within every two timestamps

Table B.6. Class Condition

Description	
A logical condition that is used to model constraints or uncertain situations.	
Hooks	
boolean check(Element e)	Defines the logical operation that checks for this condition

Table B.7. Class Variable

Description	
Simulation variables.	
Attributes	
Object value	The value of the variable

Table B.8. Class Simulation**Description**

The main simulation class, which executes a model defined by means of different structures: ResourceType, Resource, WorkGroup, Activity (TimeDrivenActivity and FlowDrivenActivity), ElementType, TimeDrivenGenerator and Flow. A simulation has an associated virtual clock that starts at startTs and advances according to the events produced by the Elements, Resources and TimeDrivenGenerators. A next-event technique is used to determine the next timestamp to advance. A minimum TimeUnit determines the accuracy of the simulation clock. The simulation ends when the virtual clock reaches endTs or no more events are available.

Attributes

String description	Description
int id	Identifier
TimeStamp startTs	Start timestamp
TimeStamp endTs	End timestamp
TimeUnit unit	Simulation time unit
Map<Integer, Resource> resourceList	Resource list
Map<Integer, Activity> activityList	Activity list
Map<Integer, ResourceType> resourceTypeList	Resource type list
Map<Integer, ElementType> elementTypeList	Element type list
Map<Integer, Flow> flowList	Workflow list
List<Generator> generatorList	Element generator list

Methods

void addInfoReceiver(InfoReceiver receiver) Adds a view to process information

Table B.8. Class Simulation – Continued

<code>void putVar(String varName, Object value)</code>	Adds a variable
<code>Variable getVar(String varName)</code>	Returns a variable
<code>void setOutput(OutputStream out)</code>	Enables debug
Hooks	
<code>void init()</code>	Before the simulation starts
<code>void end()</code>	Just after the simulation ends
<code>void beforeClockTick()</code>	Before the simulation clock advances
<code>void afterClockTick()</code>	After the simulation clock advances
Information	
<code>SimulationStartInfo</code>	Simulation start (CPU time, simulation time)
<code>SimulationEndInfo</code>	Simulation end (CPU time, simulation time)
<code>TimeChangeInfo</code>	Simulation clock advances (simulation time)

Table B.9. Class ResourceType

Description	
The type of resource. Defines roles or specialisations of the resources.	
Attributes	
<code>String description</code>	Description
<code>int id</code>	Identifier

Table B.9. Class ResourceType – Continued

Methods	
void putVar(String varName, Object value)	Adds a variable
Variable getVar(String varName)	Returns a variable
Hooks	
double beforeRoleOn()	Before a resource is made available for this resource type. An extra delay can be added before the resource is effectively made available
void afterRoleOn()	Just after a resource is made available for this resource
double beforeRoleOff()	Before a resource stops being available for this resource type. An extra delay can be added before the resource effectively stops being available
void afterRoleOff()	Just after a resource stops being available for this resource

Table B.10. Class Resource

Description
A simulation resource whose availability is controlled by means of timetable entries. A timetable entry uses a trio <ResourceType, Cycle, long> that defines a resource type, an availability cycle and the duration of each availability period. Timetable entries can overlap in time, thus allowing the resource to be potentially available for different resource types simultaneously. Cancellation entries are defined analogously to timetable entries, but instead of availability periods, define unavailability of the resource.
Attributes
String description
Description

Table B.10. Class Resource – Continued

<code>int id</code>	Identifier
<code>List<TimeTableEntry> timetable</code>	Timetable entries
Methods	
<code>void addTimeTableEntry(...)</code>	Adds a new timetable entry
<code>void addCancelTableEntry(...)</code>	Adds a new cancellation entry
<code>void putVar(String varName, Object value)</code>	Adds a variable
<code>Variable getVar(String varName)</code>	Returns a variable
Information	
<code>ResourceInfo.START</code>	Resource starts execution (resource, simulation time)
<code>ResourceInfo.END</code>	Resource ends execution (resource, simulation time)
<code>ResourceInfo.ROLON</code>	Resource is available for a specific resource type (resource, resource type, simulation time)
<code>ResourceInfo.ROLOFF</code>	Resource stops being available for a specific resource type (resource, resource type, simulation time)
<code>ResourceInfo.CANCELON</code>	Resource is cancelled for a specified period (resource, simulation time)
<code>ResourceInfo.CANCELOFF</code>	Resource is available again after a cancellation (resource, simulation time)
<code>ResourceUsageInfo.CAUGHT</code>	A resource is captured to carry out an activity (resource, resource type, activity, workflow, element, simulation time)

Table B.10. Class Resource – Continued

ResourceUsageInfo.RELEASED	A resource is freed after having been used to carry out an activity (resource, resource type, activity, workflow, element, simulation time)
----------------------------	---

Table B.11. Class WorkGroup

Description	
ResourceType[] resourceTypes int[] needed	A set of pairs <ResourceType, Integer> that defines how many resources from each type are required to do something (typically an Activity)
Attributes	
List of resource types used in this workgroup Amount of resources required per resource type. (<i>needed.length == resourceTypes.length</i>)	

Table B.12. Class TimeDrivenActivity

Description

An Activity whose finalisation is driven by a timestamp, i.e. the activity starts when there are enough resources and finishes after a period of time. This kind of activity can be characterised by a priority value, presentality, interruptibility and a set of workgroups. The workgroup selected sets how long it will take to finish this activity. By default, time-driven activities are presental, that is, an element carrying out this activity cannot simultaneously perform any other presental activity; and uninterruptible, i.e., once started, the activity keeps its resources until it is finished, even if the resources become unavailable while the activity is being performed.

Attributes

String description	Description
int id	Identifier
int priority	Priority of the activity
PrioritizedTable<ActivityWorkGroup> workGroupTable	Set of WorkGroups that the activity can use
Map<ResourceType, Long> cancellationList	Set of resource types and the period of time a resource of this type becomes unavailable after having been used for this activity
EnumSet<Modifier> modifiers	Modifiers to characterise an activity as presental or interruptible

Methods

void addWorkGroup(TimeFunction duration, WorkGroup wg, int priority, Condition cond)
priority and condition are optional arguments

Table B.12. Class TimeDrivenActivity – Continued

<code>void addResourceCancellation (ResourceType rt, double period)</code>	When a resource of the specified type (rt) is used to carry out this activity, just after the activity finishes, the resource becomes unavailable the specified period
<code>void putVar(String varName, Object value)</code>	Adds a variable
<code>Variable getVar(String varName)</code>	Returns a variable

Table B.13. Class FlowDrivenActivity

Description

An Activity that can be carried out by an Element, and whose duration depends on the finalisation of an internal Flow. This activity can be considered a hybrid Activity - StructuredFlow, its behaviour being similar to the latter.

Attributes

<code>String description</code>	Description
<code>int id</code>	Identifier
<code>int priority</code>	Priority of the activity
<code>PrioritizedTable<ActivityWorkGroup> workGroupTable</code>	Set of WorkGroups that the activity can use
<code>Map<ResourceType, Long> cancellationList</code>	Set of resource types and the period of time during which a resource of this type becomes unavailable after having been used for this activity

Table B.13. Class FlowDrivenActivity – Continued

Methods	
<code>void addWorkGroup(InitializerFlow initialFlow,</code> <code>FinalizerFlow finalFlow, WorkGroup wg, int</code> <code>priority, Condition cond)</code>	Adds a workgroup with an associated flow defined from the entry to the exit point. The priority and condition are optional arguments
<code>void addResourceCancellation (ResourceType rt,</code> <code>double period)</code>	When a resource of the specified type (rt) is used to carry out this activity, just after the activity finishes, the resource becomes unavailable for the period specified
<code>void putVar(String varName, Object value)</code>	Adds a variable
Variable <code>getVar(String varName)</code>	Returns a variable

Table B.14. Class ElementType

Description	
Describes a set of elements which have something in common. Apart from setting a priority, this is simply a descriptive attribute, and is used for statistical purposes.	
Attributes	
<code>String description</code>	Description
<code>int id</code>	Identifier
<code>int priority</code>	Priority of the elements of this type
Methods	
<code>void addElementVar(String name, Object value)</code>	Adds a variable that is individually associated with each element of this type that is created

Table B.15. Class TimeDrivenGenerator

Description	
A generator which creates elements following a time pattern.	
Attributes	
Cycle	Cycle defining when to create the elements
BasicElementCreator	Class in charge of creating elements

Table B.16. Class ElementCreator

Description	
Defines the way elements are created, that is, amount, element type to which they belong, etc.	
Attributes	
TimeFunction	How many elements are created every time
List<GenerationTrio>	The characteristics of the elements to be created, expressed as <ElementType, Flow, Probability>
Methods	
void add(ElementType, InitializerFlow, double)	Adds a new type of element to be created
Hooks	
int beforeCreateElements(int n)	Before creating the elements. <i>n</i> is the amount of elements to create, which can be modified within this method
void afterCreateElements()	Just after the elements have been created

Table B.17. Class Element

Description	
An entity capable of creating simulation events. Elements have a type and an associated Flow. Elements are not explicitly declared as part of the model, but are automatically created as described in the corresponding ElementCreator.	
Attributes	
ElementType	Type of this element
InitializerFlow	The first step of the workflow associated with this element
int id	Identifier
Information	
ElementInfo.START	Element starts execution (element, simulation time)
ElementInfo.END	Element ends execution (element, simulation time)
ElementActionInfo.REQACT	Element requests an activity (element, activity, single flow containing the activity, simulation time)
ElementActionInfo.STAACT	Element starts an activity (element, activity, single flow containing the activity, simulation time, resources captured)
ElementActionInfo.ENDACT	Element finishes an activity (element, activity, single flow containing the activity, simulation time)
ElementActionInfo.INTACT	Element interrupts an activity (element, activity, single flow containing the activity, simulation time)
ElementActionInfo.RESACT	Element resumes a previously interrupted activity (element, activity, single flow containing the activity, simulation time)

Table B.18. Class SingleFlow

Description	
The leaf node of a flow: a wrapper for activities.	
Attributes	
Activity act	The activity wrapped by this flow
Methods	
void link(Flow f)	Creates a sequence (WFPT) by linking the current flow with f
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow
void afterFinalize(Element e)	Just after the element e has finished this step
void inqueue(Element e) {}	Just after an element is enqueued in an Activity, waiting for available Resources
void afterStart(Element e) {}	Just after the element actually starts the execution of the activity

Table B.19. Class ParallelFlow

Description	
A multiple successor flow that splits the execution into several outgoing branches. Meets the <i>Parallel Split pattern</i> (WCP2).	
Methods	
void link(Flow f)	Adds an outgoing branch by linking the current flow with f

Table B.19. Class ParallelFlow – Continued

<code>void link(Collection<Flow> succlist)</code>	Adds as many outgoing branches as there are flows in <code>succlist</code>
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Table B.20. Class ExclusiveChoiceFlow

Description	
A conditional flow that allows only one of the outgoing branches, which are evaluated in order, to be activated. The flow continues via the first outgoing branch evaluated as true. Meets the <i>Exclusive Choice pattern</i> (WCP4).	
Methods	
<code>void link(Flow f)</code>	Adds a true outgoing branch by linking the current flow with <code>f</code>
<code>void link(Collection<Flow> succlist)</code>	Adds as true many outgoing branches as there are flows in <code>succlist</code>
<code>void link(Flow f, Condition cond)</code>	Adds a conditional outgoing branch by linking the current flow with <code>f</code>
<code>void link(Collection<Flow> succlist, Collection<Condition> condList)</code>	Adds as many conditional outgoing branches as there are flows in <code>succlist</code>
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Table B.21. Class MultiChoiceFlow

Description	
A conditional flow that allows all outgoing branches which meet their condition to be activated. Meets the <i>Multi-Choice pattern</i> (WCP6).	
Methods	
<code>void link(Flow f)</code>	Adds a true outgoing branch by linking the current flow with <code>f</code>
<code>void link(Collection<Flow> succList)</code>	Adds as many true outgoing branches as there are flows in <code>succList</code>
<code>void link(Flow f, Condition cond)</code>	Adds a conditional outgoing branch by linking the current flow with <code>f</code>
<code>void link(Collection<Flow> succList, Collection<Condition> condList)</code>	Adds as many conditional outgoing branches as there are flows in <code>succList</code>
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Table B.22. Class ProbabilitySelectionFlow

Description	
An adapted Exclusive Choice flow that selects one outgoing branch from among a set of them by using a probability value. Each outgoing branch has a value (0.0 - 1.0] expressing its probability of being chosen.	
Methods	
<code>void link(Flow f)</code>	Adds an outgoing branch by linking the current flow with <code>f</code> . The probability of this branch is set to 1.0.
<code>void link(Collection<Flow> succlist)</code>	Adds as many outgoing branches as there are flows in <code>succlist</code> . All branches are considered equiprobable.
<code>void link(Flow f, double prob)</code>	Adds a probabilistic outgoing branch by linking the current flow with <code>f</code>
<code>void link(Collection<Flow> succlist, Collection<Double> condlist)</code>	Adds as many probabilistic outgoing branches as there are flows in <code>succlist</code>
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Table B.23. Class SimpleMergeFlow

Description	
Creates an OR flow that allows all the incoming branches to pass. When several incoming branches arrive at the same simulation time, the outgoing branch is activated only once. Meets the <i>Simple Merge pattern</i> (WCP5).	
Methods	
void link(Flow f)	Creates a sequence (WFPT) by linking the current flow with f
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow

Table B.24. Class MultiMergeFlow

Description	
Creates an OR flow that allows all the incoming branches to pass. Meets the <i>Multi-Merge pattern</i> (WCP8).	
Methods	
void link(Flow f)	Creates a sequence (WFPT) by linking the current flow with f
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow

Table B.25. Class SynchronizationFlow

Description	An AND join flow that passes only when all the incoming branches have been activated once. Meets the Synchronisation pattern (WCP3).
Methods	<code>void link(Flow f)</code> Creates a sequence (WFP1) by linking the current flow with <code>f</code>
Hooks	<code>boolean beforeRequest(Element e)</code> Before the element <code>e</code> requests this flow

Table B.26. Class DiscriminatorFlow

Description	An AND join flow that allows only the first incoming branch to pass. Meets the <i>Blocking Discriminator pattern</i> (WCP28).
Methods	<code>void link(Flow f)</code> Creates a sequence (WFP1) by linking the current flow with <code>f</code>
Hooks	<code>boolean beforeRequest(Element e)</code> Before the element <code>e</code> requests this flow

Table B.27. Class PartialJoin

Description	
An AND join flow that allows only the n-th incoming branch to pass. Meets the <i>Blocking Partial Join pattern</i> (WCP31).	
Methods	
<code>void link(Flow f)</code>	Creates a sequence (WFPI) by linking the current flow with <code>f</code>
Attributes	
<code>int acceptValue</code>	Number of incoming branches which activate the flow
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Table B.28. Class InterleavedParallelRoutingFlow

Description	
A structured flow that contains a set of activities to be performed according to a predefined set of partial orderings. Partial orderings are defined using a collection of activity arrays. Each array [A1, A2, ... An] defines precedence relations; thus, A1 must be executed before A2, A2 before A3, and so on. If all the activities are presental, meets the <i>Interleaved Parallel Routing pattern</i> (WCP17)	
Attributes	
<code>Collection<Activity> activities</code>	The activities executed within this flow.

Table B.28. Class InterleavedParallelRoutingFlow – Continued

Collection<Activity[]> dependencies	The partial orderings defined for this structure. Partial orderings are defined using a collection of arrays. Each array [A1, A2, ... An] defines precedence relations; thus, A1 must be executed before A2, A2 before A3, and so on.
Methods	
void link(Flow f)	Creates a sequence (WFPI) by linking the current flow with f
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow
void afterFinalize(Element e)	Just after the element e has finished this step

Table B.29. Class DoWhileFlow

Description	
A do-while loop. The inner flow is executed the first time and then the post-condition is checked. If the post-condition is true, the inner flow is executed again; otherwise, this flow finishes. Meets the <i>Structures Loop pattern</i> (WCP21).	
Attributes	
Condition cond	Post-condition that controls the loop operation
Methods	
void link(Flow f)	Creates a sequence (WFPI) by linking the current flow with f

Table B.29. Class DoWhileFlow – Continued

<code>void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)</code>	Sets the inner flow of the loop to start with <code>initialBranch</code> and end at <code>finalBranch</code>
<code>void addBranch(Flow branch)</code>	Sets the inner flow of the loop to be a single flow
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow
<code>void afterFinalize(Element e)</code>	Just after the element <code>e</code> has finished this step

Table B.30. Class WhileDoFlow

Description	
A while-do loop. A precondition is checked before executing the inner flow. If the precondition is false, this flow finishes. Meets the <i>Structures Loop pattern</i> (WCP21).	
Attributes	
Condition <code>cond</code>	Precondition that controls the loop operation
Methods	
<code>void link(Flow f)</code>	Creates a sequence (WFPT) by linking the current flow with <code>f</code>
<code>void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)</code>	Sets the inner flow of the loop to start with <code>initialBranch</code> and end at <code>finalBranch</code>
<code>void addBranch(Flow branch)</code>	Sets the inner flow of the loop to be a single flow
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Table B.30. Class WhileDoFlow – Continued

void afterFinalize(Element e)
 Just after the element e has finished this step

Table B.31. Class ForLoopFlow

Description	
A for loop. The inner flow is executed N times. Meets the <i>Structures Loop pattern</i> (WCP21).	
Attributes	
TimeFunction iterations	The number of loop iterations
Methods	
void link(Flow f)	Creates a sequence (WFP1) by linking the current flow with f
void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)	Sets the inner flow of the loop to start with initialBranch and end at finalBranch
void addBranch(Flow branch)	Sets the inner flow of the loop to be a single flow
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow
void afterFinalize(Element e)	Just after the element e has finished this step

Table B.32. Class `StaticPartialJoinMultipleInstancesFlow`

Description	
	A structured flow that meets the <i>Static Partial Join for Multiple Instances pattern</i> (WCP34) if the acceptance value is greater than 1 and less than the number of thread instances created within this structure. If both the acceptance value and the number of instances are the same, meets the <i>Multiple Instances with a Priori Design-Time Knowledge pattern</i> (WCP13) (the <code>SynchronizedMultipleInstanceFlow</code> class can be used for the same purpose).
Attributes	
<code>int nInstances</code>	The number of thread instances created within this structure
<code>int acceptValue</code>	The number of thread instances that must reach the end of the structure to continue execution
Methods	
<code>void link(Flow f)</code>	Creates a sequence (WFPT) by linking the current flow with <code>f</code>
<code>void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)</code>	Sets the inner flow of the structure to start with <code>initialBranch</code> and end at <code>finalBranch</code>
<code>void addBranch(Flow branch)</code>	Sets the inner flow of the structure to be a single flow
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow
<code>void afterFinalize(Element e)</code>	Just after the element <code>e</code> has finished this step

Table B.33. Class StructuredPartialJoinFlow

Description	
A structured flow whose initial step is a parallel flow and whose final step is a partial join flow. Meets the <i>Structured Partial Join pattern</i> (WCP30).	
Attributes	
<code>int acceptValue</code>	The number of branches that must be finished in order to continue execution
Methods	
<code>void link(Flow f)</code>	Creates a sequence (WFPT) by linking the current flow with <code>f</code>
<code>void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)</code>	Adds an inner flow to the structure that starts with <code>initialBranch</code> and ends at <code>finalBranch</code>
<code>void addBranch(Flow branch)</code>	Adds an inner flow to the structure that is a single flow
Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow
<code>void afterFinalize(Element e)</code>	Just after the element <code>e</code> has finished this step

Table B.34. Class StructuredSynchroMergeFlow

Description	
A structured flow whose initial step is a multi-choice flow and whose final step is a synchronisation. Meets the <i>Structured Synchronisation pattern</i> (WCP7).	
Methods	
void link(Flow f)	Creates a sequence (WFPI) by linking the current flow with f
void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch, Condition cond)	Adds a branch, which is executed under certain conditions, connecting the predefined entry and exit of this flow
void addBranch(TaskFlow branch, Condition cond)	Adds a branch, which is executed under certain conditions, comprising a single flow
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow
void afterFinalize(Element e)	Just after the element e has finished this step

Table B.35. Class InterleavedRoutingFlow

Description	
A structured flow whose initial step is a parallel flow and whose final step is a synchronisation flow. Meets the <i>Interleaved Routing pattern</i> (WCP40) if all the activities are presental.	
Methods	
void link(Flow f)	Creates a sequence (WFPI) by linking the current flow with f

Table B.35. Class InterleavedRoutingFlow – Continued

void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)	Adds an inner flow to the structure that starts with initialBranch and ends at finalBranch
void addBranch(Flow branch)	Adds an inner flow to the structure that is a single flow
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow
void afterFinalize(Element e)	Just after the element e has finished this step

Table B.36. Class StructuredDiscriminatorFlow

Description	
A structured flow whose initial step is a parallel flow and whose final step is a discriminator flow. Meets the <i>Structured Discriminator pattern</i> (WCP9).	
Methods	
void link(Flow f)	Creates a sequence (WFPI) by linking the current flow with f
void addBranch(InitializerFlow initialBranch, FinalizerFlow finalBranch)	Adds an inner flow to the structure that starts with initialBranch and ends at finalBranch
void addBranch(Flow branch)	Adds an inner flow to the structure that is a single flow
Hooks	
boolean beforeRequest(Element e)	Before the element e requests this flow
void afterFinalize(Element e)	Right after the element e has finished this step

Table B.37. Class ThreadSplitFlow

Description	
A flow that creates several instances of the outgoing branch. It should be used with its counterpart ThreadMergeFlow. Meets the <i>Thread Split pattern</i> (WCP 42).	
Attributes	
int nInstances	Amount of instances of the outgoing branch that are created
Methods	
void link(Flow f)	Creates a sequence (WFPI) by linking the current flow with f
Hooks	
boolean beforeRequest (Element e)	Before the element e requests this flow

Table B.38. Class ThreadMergeFlow

Description	
A flow that merges a specified number of instances. It should be used with its counterpart, the ThreadSplitFlow. Meets the <i>Thread Merge pattern</i> (WCP 41), but also has extra features. Works as a thread discriminator if acceptValue is set to 1; or as a thread partial join if any other value greater than one and lower than nInstances is used.	
Attributes	
int nInstances	The number of threads expected before this node is reset
int acceptValue	The number of threads expected to activate the merge

Table B.38. Class ThreadMergeFlow – Continued

Methods	
<code>void link(Flow f)</code>	Creates a sequence (WFPI) by linking the current flow with <code>f</code>

Hooks	
<code>boolean beforeRequest(Element e)</code>	Before the element <code>e</code> requests this flow

Other Java DES Tools

There is a large collection of Java-based software for Discrete-Event System Simulation (DES). A few examples, which can currently be downloaded and used, are:

- JSIM (Miller et al., 1997)
- simJava (Howell and McNab, 1998)
- Simkit (Buss, 2002)
- javaSimulation (Helsgaun, 2004)
- PsimJ (Garrido and Im, 2004)
- SSJ (L'ecuyer and Buist, 2005)
- DESMO-J (Page and Kreutzer, 2005)
- JAPROSIM (Abdelhabib and Brahim, 2008)

This list expands more if ad-hoc solutions for specific problems are included. This appendix reviews some of the pre-eminent Java-based DES tools.

C.1. JSIM

JSIM (Miller et al., 1997, 2000) is one of the first attempts to develop a discrete event simulator with Java. Like many other similar tools, reducing the user's coding effort is its final aim. The major assets of this tool are flexibility and the stress on graphical and animation capabilities, all of them intended to reinforce the use of the tool on the web. Both event and process orientation are supported.

Two formalisms support JSIM: graph theory and query-driven simulation (Miller et al., 1991). The main idea behind the use of this tool is having a database with simulation models and results. Users wishing to know the outcome of simulating a certain model would obtain their answer transparently either by looking for previous simulation executions (thus saving the users from having to wait for the simulation to be executed) or by launching a new simulation.

JSIM consists of five basic packages: *queue*, which offers different queue structures; *statistic*, which collects and analyses data from the simulation; *variate*, which includes random number generators; *process*, which implements process-oriented models; and *event*, which implements event-oriented models.

Table C.1. Summary JSIM

Name	JSIM
URL	http://www.cs.uga.edu/~jam/jsim/
Last version available	1.4 (2007)
Licence	BSD
Source code	Yes
Graphical interface	Yes
Statistics	Yes
Manuals	No (only installation support)
Related publications	9
Most recent publication	2001

C.2. Simkit

Simkit (Buss, 2002), like JSIM, is based on graph theory, more specifically, on the Event Graph formalism (Schruben, 1983). An event graph maps events onto nodes, and edge weights indicate the time required to schedule the next event. Edges can contain conditions as well, which are used to determine whether or not an event must be executed. The use of event graphs makes Simkit an event-oriented tool. Any other worldview is not directly supported.

Another interesting aspect of Simkit is the use of the *Listener* software design pattern (Gamma et al., 1995). By using such a pattern, not only can the simulation be implemented as a set of loosely-coupled components, but changes in the system state variables can be observed more easily.

Table C.2. Summary Simkit

Name	Simkit
URL	http://diana.nps.edu/Simkit/
Last version available	1.3.8 (4th, January 2009)
Licence	LGLP
Source code	Yes
Graphical interface	-
Statistics	Yes
Manuals	Yes (lab examples)
Related publications	5
Most recent publication	2005

C.3. SSJ

SSJ is a Java library for stochastic simulation. It provides facilities for generating uniform and nonuniform random variates, computing different measures related to probability distributions, performing goodness-of-fit tests, applying

quasi-Monte Carlo methods, collecting statistics, and programming discrete-event simulations with both event-oriented and process-oriented approaches.

The library comprises the following packages:

- *probdist* Probability distributions.
- *gof* Univariate goodness-of-fit statistical tests, such as Kolmogorov-Smirnov, Chi-square and Anderson-Darling.
- *rng* Uniform Random number generation.
- *hups* Highly uniform point sets and sequences (HUPS) over the s -dimensional unit hypercube $[0, 1)^s$, and tools for their randomisation. These techniques are used for quasi-Monte Carlo (QMC) and randomised QMC numerical integration.
- *randvar* Non-uniform random variate generation, primarily from standard distributions.
- *stat* Basic tools for collecting statistics and computing confidence intervals. Most classes implement the observer software design pattern.
- *simevents* Classes for handling event scheduling simulation, including a simulation clock and different implementations of the event list.
- *simprocs* Classes for handling process interaction simulation, including processes themselves, resources, bins and conditions.

C.4. DESMO-J

DESMO-J (Page and Kreutzer, 2005) is an object-oriented framework that extends Java to provide the modeller with different tools to make building a DES model easier. Such tools include:

Table C.3. Summary SSJ

Name	SSJ
URL	http://www.iro.umontreal.ca/~simardr/ssj/indexe.html
Last version available	2.4 (8th September 2010)
Licence	GPL
Source code	Yes
Graphical interface	Yes (only for displaying results)
Statistics	Yes
Manuals	Yes
Related publications	4
Most recent publication	2005

- Several classes that represent common modelling components, such as queues, random number generators, data collection utilities, etc.
- Abstract classes that can be adapted to create models, entities, events and processes with customised behaviour. The user can select either events and entities (for an event-oriented simulation) or processes (for a process-oriented simulation).
- An “Experiment” class comprising a scheduler, event list and simulation clock, as well as tools for report generation and tracing simulation runs.

This design is intended to isolate the model from the experiment, thus supporting the execution of different models within the same experiment for the purpose of evaluating several scenarios, and also of different experiments within the same model.

DESMO-J consists of the following packages:

- *desmoj.core* Basic modelling classes.

- *desmoj.core.simulator* Basic classes for model building and experiment preparation. This package also includes modelling components such as queues, entities, events and processes.
- *desmoj.core.dist* Probability distributions based on the linear congruential random number generator by Donald Knuth.
- *desmoj.core.exception* Internal exceptions.
- *desmoj.core.report* Automated report generation (HTML and XML).
- *desmoj.core.statistic* Simulation data collectors: counters, averages, histograms... All the statistics are displayed in the final report.
- *desmoj.core.advancedModellingFeatures* Process-oriented modelling classes with a higher abstraction level.
- *desmoj.extensions.applications* Extensions to build models in different application fields.
- *desmoj.extensions.experimentation* Classes to define a basic graphical interface to execute experiments and view results.

C.5. JAPROSIM

This library (Abdelhabib and Brahim, 2008) is clearly inspired by the concepts of the foundational language SIMULA. Hence, active entities, that is, transient entities that move through the system, are used to describe the system's behaviour. Active entities can be seen as processes in the process interaction worldview.

JAPROSIM includes the following packages:

- *kernel* Active entities, scheduler, queues and resources.
- *random* Random number generation.

Table C.4. Summary DESMO-J

Name	DESMO-J
URL	http://desmoj.sourceforge.net/home.html
Last version available	2.2.0 (June 2010)
Licence	Apache
Source code	Yes
Graphical interface	Yes (only for displaying results)
Statistics	Yes
Manuals	Yes
Related publications	1 book
Most recent publication	2009

- *distributions* Probability distributions.
- *statistics* Statistical variables.
- *gui* Graphical user interface classes to use for project parameterisation, trace and simulation result presentation.
- *utilities* Classes for model development.

The authors claim that

« It (*JAPROSIM*) has the major key future of automatic and implicit collection of statistics over other similar frameworks. »

Only Simkit offers something similar, but it remains unclear why this feature makes a difference with tools like Desmo-J, where the statistics package has to be explicitly used. Moreover, the memory and CPU costs of collecting *everything* by default are evident if all of these data are not specifically required by the simulation. Insisting on efficiency, the tool uses a direct translation from active entities into Java threads, whose dangers were already discussed in Section 2.2.

Table C.5. Summary JAPROSIM

Name	JAPROSIM
URL	http://sourceforge.net/projects/japrosim/
Last version available	- (10th January 2007)
Licence	-
Source code	Yes
Graphical interface	Yes (only for defining experiment parameters)
Statistics	Yes
Manuals	Yes
Related publications	1
Most recent publication	2008

Other Java PDES Tools

Java-based Parallel Discrete Event Simulation (PDES) are not as numerous as Java-based DES. Indeed, only a few examples can be considered as *generic* simulators. We will review the most significant Java PDES tools in the following sections.

D.1. JUST: Java Ubiquitous Simulation Tools

Cassel and Pidd (2001) propose a distributed DES based on the three-phase approach (see Subsection 1.2.4).

The initial version of the tool was a client-server package: the server managed the simulation and the clients implemented the model.

A second version, JUSTDistributed, is a PDES with conservative synchronisation, featuring null message and lookahead. JUSTDistributed is multithreaded and uses RMI for communication. The way the authors deal with the resource contention and shared states problems will be explained later in Section 4.2.

D.2. SPADES/Java

SPaDES /Java is a PDES library, intended to isolate the user from the implementation details involving event synchronisation and parallelisation (Teo and Ng,

2002).

This process-oriented simulator supports both sequential and parallel execution. The latter uses a conservative null-message protocol for synchronisation.

When modelling with the library, the conceptual model consists of permanent and temporary processes. Resources are treated as permanent processes and modelled as LPs, whereas any other entity is treated as a temporary process and modelled as a time-stamped message passed between LPs. Actually, only permanent processes are implemented as Java threads.

Programming a parallel simulator with SPaDES/Java requires using a template comprising four main parts: process definitions, process routines that define the simulation logic, code for initialising and starting the simulator, and a message abstraction and reconstruction routine for parallel simulations.

D.3. D-SOL

Jacobs et al. (2002) presents a fully distributed simulation environment called D-SOL. This tool is an example of a dedicated execution simulator.

The simulation framework consists of two remote network services: an event-based DES remote simulation service and a system representation service, which includes representation and statistical libraries. Such service-oriented architecture is intended to enhance flexibility, distribution of simulation models, and integration and interaction with real-time information systems.

The simulation library core is event-based, since the authors insist on the advantages of this world-view over the activity scanning and the process interaction approaches (see Section 1.2). However, a porting of the process interaction approach to be supported by the event-based core is presented (Jacobs and Verbraeck, 2004), as already explained in Section 2.4.

The last versions of D-SOL, as explained in (Jacobs, 2005), include multiformalism support. Thus, the simulation framework allows a user to base its development on the Discrete Event System Specification (DEVS), Differential equa-

tion system specification (DESS) or mixed DEVS-DESS formalisms (Zeigler et al., 2000).

D.4. CSA&S/PV: Complex Systems Analysis & Simulation - Parallel Version

Niewiadomska-Szynkiewicz et al. (2003) have developed a parallel software environment based on an earlier design programmed in C. This tool is intended to aid a user when designing and simulating complex physical processes.

The parallelism is exploited by using a conservative LP approach, but the user can configure each LP local clock to be either event- or time-driven (i.e., a clock which advances at regular intervals). The tool developed consists of five modules:

- Shell. The user graphical interface.
- Calculation module or manager: The system kernel, which handles computation and interprocess communication.
- Communication library: An interface that provides communication facilities between the shell and the calculation module.
- User application: The simulation model as defined by the user. The LPs are defined in this module.
- User library: A set of methods to communicate the user application and the manager.

CSA&S/PV has been applied mainly to computer and water networks, but it is intended to be a general purpose simulation tool.

D.5. Summary of Java PDES Tools

Table D.1 summarises the characteristics of the tools mentioned earlier in this section, plus further parallel simulators: JTED (Cowie, 1998), Fornax (Halderen and Overeinder, 1998), JWarp (Bizarro et al., 1998). Most of these simulators exploit spatial domain decomposition, that is, the classical LP approach. Only D-SOL follows a different path and focuses on dedicated execution. The most (negatively) noticeable aspect is that, out of the tools presented here, only D-SOL is currently available for free download and use. Actually, the SPaDES/-Java website is online, but the form for requesting the software does not work correctly, and we have not been able to download the simulator.

Table D.1. Summary of Java PDES tools

PDES tool	Parallelism exploitation	Worldview
JUST	Domain Decomposition (Conservative)	Three-Phase
SPaDES/Java	Domain Decomposition (Conservative)	Process-oriented
D-SOL	Dedicated Execution	Multiformalism
CSA&S/PV	Domain Decomposition (Conservative)	Event-based
JTED	Domain Decomposition (Conservative)	Event-based
Fornax	Domain Decomposition (Optimistic and Conservative)	Process-oriented
JWarp	Domain Decomposition (Optimistic)	Event-based

The unavailability of the Java PDES tools listed limits not only possible comparisons, but the opportunity to benefit from solutions that have already been

developed. Furthermore, the goals of D-SOL, the only tool available, differ greatly from those listed in Table 4.1. D-SOL is intended to be distributed, in the sense of being executed in a loosely coupled system, whereas PSIGHOS tries to exploit parallelism in tightly coupled systems, such as multi-core computers. Hence, when being executed in a distributed environment, D-SOL's main concern is not reducing the execution time, but offering the final user the possibility to easily access and reuse models and results.

Workflow Patterns

The 43 control-flow patterns proposed by Russell et al. (2006) (numbered WCP1 - WCP43) delineate the fundamental requirements for modelling the different scenarios defined within this perspective. These patterns are based on a study of countless practical cases involving real companies, and set both a language- and technology- independent framework to describe, analyse and compare different workflow languages and tools.

The rest of this appendix will review the main categories these patterns are organised into. For a comprehensive review on these patterns, (Workflow Patterns Initiative, 2010) offers the most updated information and interactive contents.

Table E.1. Basic control flow patterns

Id	Pattern	Description
WCP1	Sequence	A task in a workflow is enabled after the completion of a preceding task in the same process
WCP2	Parallel Split	The divergence of a branch into two or more parallel branches each of which executes concurrently
WCP3	Synchronisation	The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled
WCP4	Exclusive Choice	The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical expression associated with the branch
WCP5	Simple Merge	The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch

Table E.2. Advanced branching and synchronisation patterns

Id	Pattern	Description
WCP6	Multi-Choice	The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is passed to one or more of the outgoing branches based on the outcome of distinct logical expressions associated with each of the branches
WCP7	Structured Synchronising Merge	The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled
WCP8	Multi-Merge	The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch
WCP9	Structured Discriminator	The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablement of incoming branches do not result in the thread of control being passed on. The discriminator construct resets when all incoming branches have been enabled

Table E.2 – Continued

Id	Pattern	Discrimina-	Description
WCP28	Blocking Discriminator		The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablement of incoming branches are blocked until the discriminator has reset
WCP29	Cancelling Discriminator	Discrimi-	The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the Cancelling Discriminator also cancels the execution of all of the other incoming branches and resets the construct
WCP30	Structured Partial Join		The convergence of M branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. Subsequent enablement of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled

Table E.2 – Continued

Id	Pattern	Description
WCP31	Blocking Partial Join	The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled. The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablement of incoming branches are blocked until the join has reset
WCP32	Cancelling Partial Join	The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct
WCP33	Generalised AND-Join	The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings

Table E.2 – Continued

Id	Pattern	Description
WCP37	Local Synchronising Merge	The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronisation is made on the basis of information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge
WCP38	General Synchronising Merge	The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time
WCP41	Thread Merge	At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution
WCP42	Thread Split	At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance

Table E.3. Multiple instance patterns

Id	Pattern	Description
WCP12	Multiple Instances without Synchronisation	Within a given process instance, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronise them upon completion
WCP13	Multiple Instances with a Priori Design-Time Knowledge	Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronise the activity instances at completion before any subsequent activities can be triggered
WCP14	Multiple Instances with a Priori Run-Time Knowledge	Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronise the instances at completion before any subsequent tasks can be triggered

Table E.3 – Continued

Id	Pattern	Description
WCP15	Multiple Instances without a Priori Run-Time Knowledge	Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronise the instances at completion before any subsequent tasks can be triggered
WCP34	Static Partial Join for Multiple Instances	Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered. Subsequent completions of the remaining M-N instances are inconsequential
WCP35	Cancelling Partial Join for Multiple Instances	Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered and the remaining m-n instances are cancelled

Table E.3 – Continued

Id	Pattern	Description
WCP36	Dynamic Partial Join for Multiple Instances	<p>Within a given process instance, multiple concurrent instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so had not been disabled. A completion condition is specified which is evaluated each time an instance of the task completes. Once the completion condition evaluates to true, the next task in the process is triggered. Subsequent completions of the remaining task instances are inconsequential and no new instances can be created</p>

Table E.4. State-based patterns

Id	Pattern	Description
WCP16	Deferred Choice	<p>A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn</p>

Table E.4 – Continued

Id	Pattern	Description
WCP17	Interleaved Parallel Routing	A set of activities has a partial ordering defining the requirements with respect to the order in which they must be executed. Each activity in the set must be executed once and they can be completed in any order that accords with the partial order. However, as an additional requirement, no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time)
WCP18	Milestone	A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger
WCP39	Critical Section	Two or more connected subgraphs of a process model are identified as “critical sections”. At runtime for a given process instance, only tasks in one of these “critical sections” can be active at any given time. Once execution of the tasks in one “critical section” commences, it must complete before another “critical section” can commence

Table E.4 – Continued

Id	Pattern	Description
WCP40	Interleaved Routing	Each member of a set of activities must be executed once. They can be executed in any order but no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time). Once all of the activities have completed, the next activity in the process can be initiated

Table E.5. Cancellation and force completion patterns

Id	Pattern	Description
WCP19	Cancel Task	An enabled task is withdrawn prior to it commencing execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed
WCP20	Cancel Case	A complete process instance is removed. This includes currently executing tasks, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully
WCP25	Cancel Region	The ability to disable a set of tasks in a process instance. If any of the tasks are already executing (or are currently enabled), then they are withdrawn. The tasks need not be a connected subset of the overall process model

Table E.5 – Continued

Id	Pattern	Description
WCP26	Cancel Multiple Instance Activity	Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. At any time, the multiple instance task can be cancelled and any instances which have not completed are withdrawn. Task instances that have already completed are unaffected
WCP27	Complete Multiple Instance Activity	Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronise the instances at completion before any subsequent tasks can be triggered. During the course of execution, it is possible that the task needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent tasks

Table E.6. Iteration patterns

Id	Pattern	Description
WCP10	Arbitrary Cycles	The ability to represent cycles in a process model that have more than one entry or exit point

Table E.6 – Continued

Id	Pattern	Description
WCP21	Structured Loop	The ability to execute an activity or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point
WCP22	Recursion	The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated

Table E.7. Termination patterns

Id	Pattern	Description
WCP11	Implicit Termination	A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successfully completed
WCP43	Explicit Termination	A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is cancelled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed

Table E.8. Trigger patterns

Id	Pattern	Description
WCP23	Transient Trigger	The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilised if there is a task instance waiting for it at the time it is received
WCP24	Persistent Trigger	The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task

Computational Concurrency

Concurrency is a property of systems in which several computational processes¹ are executing at the same time, and potentially interacting with each other. The concurrent processes may be executing truly simultaneously, such as when they are run on separate processors, or their execution steps may be interleaved to produce the appearance of simultaneousness, as in the case of separate processes running on a multitasking system. Because the processes in a concurrent system can interact with each other while they are executing, the number of possible execution paths in the system can be extremely large, and the resulting behaviour can be very complex (Roscoe et al., 1997).

F.1. Awareness Level in the Interaction Between Processes

It is possible to classify the interactions between processes based on the level of awareness each process has of the existence of the others (Burns and Davies, 1993). There are three levels of awareness:

¹From the point of view of Operating Systems theory, there is a clear difference between *process* and *thread*. A process is an abstraction of a program in execution whose main objective is to group related resources; whereas a *thread* or light process is a piece of a process that can be executed. For the sake of simplicity, we will assume both concepts as synonymous. A comprehensive review on this topic can be found in (Tanenbaum, 2007).

The processes are unaware of each other The processes are not designed to operate together; hence, the results of a process are independent of the other processes' actions. The Controlling System has to manage any competition for the available resources.

The processes are indirectly aware of each other Although the processes are not directly aware of each other, they share access to some system objects, meaning the results of a process may depend on information obtained from others. Cooperation is thus established to share the common object. This is the least common level of knowledge, but the most involved because the information has to be kept updated. Each process stores a copy of the information to allow for simultaneous access. These copies also allow for changes to be written locally and let the underlying system update all the other copies so as to keep the data consistent.

The processes are directly aware of each other The processes are able to communicate directly with each other and are designed to work together on activities. These processes exhibit cooperative communication such that the results of one process can depend on information obtained from others.

F.2. Competition between Processes for Resources

Concurrent processes conflict with each other when competing for the same resource. The state of said resource should remain unaltered after being used by a process. Even if there is no exchange of information between competing processes, the execution of a process could influence the behaviour of said processes. Specifically, if two processes wish to access the same resource, the controlling system must resolve and control the access to the resource among the competing processes such that one will be assigned the resource while the other is forced to wait. The process that is denied access is blocked and will be delayed. In a worst case scenario, this means the blocked process will not gain access to the

resource and will not finish its task successfully. Three control problems have to be addressed when faced with competing processes:

Mutual exclusion Suppose two processes wish to access a single, non-shareable resource. These so-called critical processes are used by that part of the program known as the critical section. It is important that only one process gain access to the critical section at any given time. Solving this mutual exclusion gives rise to the two following problems.

Deadlock Consider two processes, P_1 and P_2 , and critical resources R_1 and R_2 . Suppose each process needs to access both resources to carry out part of its function. In this case, the system controller may assign R_1 to P_2 and R_2 to P_1 . Each process is waiting on one of the two resources and neither will relinquish its own process until it can access the other and execute its critical section. Both processes are deadlocked. Another situation leading to deadlock is when the processes are input-locked, waiting for a message that is never going to arrive.

Starvation This occurs when a process waits indefinitely for a resource. This is solved by assigning a higher priority to processes to grant them quicker access to the resource being requested, thus avoiding starvation (aging).

F.3. Cooperation between Processes by Sharing

Several processes can have access to shared resources. The processes can use and update the data associated with these shared resources without referencing the other processes, all the while being aware that said processes can access the same data. The processes thus have to cooperate to ensure that the shared data are handled correctly. The control mechanisms have to ensure the integrity of the data. The only difference with the access described in the preceding section is that the data may be accessed in two different ways, one to read and the other to write. Only write operations have to be mutually exclusive. Before these pro-

blems are addressed, however, a new requirement must be introduced: data coherence. In this case, each process's complete sequence can be labelled as critical to ensure the integrity of the data, even if no critical resources are involved.

F.4. Control Mechanisms

Both cooperation and competition require a series of control mechanisms, some of which are briefly described below. For an in-depth review of these structures, there are many books on computational concurrency such as (Andrews, 2000).

F.4.1. Locks

A lock is the most basic mechanism for synchronising processes. A lock can be acquired at any given moment by one or by no processes. If a process P_A tries to acquire a lock already acquired by another process P_B , P_A will stop running until P_B releases the lock.

Locks are normally used to synchronise access to a shared resource, each of which has an associated Lock object. Considered pseudo-code, there are two main methods for handling locks:

- The *acquire* method takes the lock, allowing the process to access the resource. Upon calling the method, the process is forced to wait for the resource if necessary.
- The *release* method frees the lock.

Listing F.1 shows the use of these methods.

Listing F.1. Use of Lock access methods

```
lock = new Lock();  
lock.acquire(); // will block if lock is already held  
// ... access shared resource  
lock.release();
```

F.4.2. Re-entrant locks (RLock)

A re-entrant lock (RLock) is a version of the Lock that only blocks if the mechanism has already been acquired by another process. Unlike the simple Lock, RLock does not block if the same process tries to acquire it twice, as shown in Listing F.2.

Listing F.2. Comparison between Lock and RLock acquisition

```
lock = new Lock();
lock.acquire();
lock.acquire(); // this will block

lock = new RLock();
lock.acquire();
lock.acquire(); // this will not block
```

The main use of RLock is with nested access to shared resources. Since this mechanism stores the recursion level, the number of release and acquire calls has to be balanced.

F.4.3. Monitor

The definition of monitor was first proposed by Hansen (1973) and later refined by Hoare (1974). Monitors provide a structured mechanism that ensures exclusive access to resources, while easing synchronisation and communication between different processes. The monitor achieves this by encapsulating both the resource definition as well as a series of operations for its exclusive manipulation. Only one of the calls to these operations can be active at once, thus protecting the interior of the monitor from being accessed simultaneously by multiple processes. This achieves exclusively mutual access that blocks and queues those processes trying to access a busy monitor. These tasks are synchronised within the monitor through the use of conditional variables that allow for a process being executed by the monitor to be delayed. Two operations, wait and signal, are associated with each condition. The wait operation is used to block/suspend

the execution of the process invoking it if the condition is true. When the process is blocked, another process is allowed to use the monitor. When the same condition becomes false, a signal call continues the execution of one of the processes suspended after a wait. If there are several processes, only one continues; if there are no processes, the signal call is ignored. These operations allow for multiple processes to be in a monitor at once, although only one will be active.

F.4.4. Barrier Synchronisation

Many problems can be solved using iterative algorithms. In some cases, the operations to be performed during an iteration or *episode* can be distributed among several processes and executed in parallel. Since the results of an episode generally depend on the results of the previous episode, a synchronisation mechanism is required that prevents a new episode from starting before the previous one has been completed.

A *barrier* constitutes a point where a process arrives and waits until the remaining processes have reached the same point. Listing F.3 shows the typical use of a barrier.

Listing F.3. Typical algorithm with a barrier

```
Process worker {
    while (true) {
        ... // Performs work of the current iteration
        barrier.await(); // Waits for all the workers to complete
                        work
    }
}
```

A barrier can be implemented in many ways, two popular structures being centralised and tree-based. A centralised barrier uses a shared variable (usually a simple counter) that keeps track of the processes as they reach the barrier. Obviously, the use of a single shared variable evolves into a severe memory contention. Hence, for a large number of processors, tree-based barriers represent a

reasonable alternative. Tree-based barrier utilises a set of distributed flags that are combined in successive stages, thus reducing the contention. Butterfly, dissemination and tournament are different examples of tree-based barriers.

A user can profit from barriers by performing global operations (a sum, computing the minimum value...) with all the processes during a synchronisation. Well-known parallel languages, such as MPI (Snir et al., 1998), refer to these operations as global *reductions*.

Bibliography

The following references are author-ordered.

- W. V. D. Aalst, J. Desel, and E. Kindler. On the semantics of epcs: A vicious circle. In *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80, 2002.
- B. Abdelhabib and B. Brahim. JAPROSIM: A Java framework for Process Interaction Discrete Event Simulation. *Journal of Object Technology*, 7(1):103–119, 2008.
- A. Agarwal and M. Hybinette. Merging Parallel Simulation Programs. *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, pages 227–233, 2005. doi: 10.1109/PADS.2005.10. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1443328>.
- R. M. Aguilar. *Aportaciones Metodológicas Basadas en Simulación e Inteligencia Artificial para la Toma de Decisiones en la Gerencia Hospitalaria*. Phd thesis, Universidad de La Laguna, 1998.
- R. M. Aguilar, I. Castilla, V. Muñoz, J. I. Estévez, C. A. Martín, and M. L. Predictive Simulation for Multiagent Resource Distribution: An Application in Hospital Management. In *17th European Modeling & Simulation Symposium proceedings (vol. I)*, Marseille, France, 2005a.

- R. M. Aguilar, C. A. Martín, I. Castilla, V. Muñoz, and L. Moreno. Librería Java para la Simulación de Sistemas de Eventos Discretos y su Aplicación en la Gerencia Hospitalaria. *Revista Iberoamericana de Automática e Informática Industrial*, 2(4):66–77, 2005b.
- R. M. Aguilar, I. Castilla, R. Muñoz, C. A. Martín, and J. Piñeiro. Verification and validation in discrete event simulation: A case study in hospital management. In *International Mediterranean Modeling Multiconference. EMSS 2006*, 2006.
- R. M. Aguilar, I. Castilla, and R. Muñoz. Hospital Resource Management. In *Simulation-Based Case Studies in Logistics. Education and Applied Research*, pages 65–84. Springer-Verlag, 2009.
- M. Alla. *Simulation applications in health care*. Master thesis, Riga Technical University, 2005.
- G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, Reading, Massachusetts, 2000. ISBN 0-201-35752-6.
- A. W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.
- O. Balci. The implementation of four conceptual frameworks for simulation modeling in high-level languages. In *Proceedings of the 20th conference on Winter simulation - WSC '88*, pages 287–295, New York, 1988. ACM Press. ISBN 0911801421. doi: 10.1145/318123.318204. URL <http://portal.acm.org/citation.cfm?doid=318123.318204>.
- C. Ball and M. Bull. Barrier Synchronisation in Java. Technical report, UK High-End Computing, 2003. URL http://www.ukhec.ac.uk/publications/reports/synch_java.pdf.
- U. Banerjee, R. Eigenmann, a. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993. ISSN 00189219.

- doi: 10.1109/5.214548. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=214548>.
- J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, Inc, New Jersey, 3rd edition, 2000.
- P. J. Baquero, R. M. Aguilar, A. Ayala, and I. Castilla. Simulation of an IT User Support Center. In *17th European Modeling & Simulation Symposium proceedings (vol. I)*, Marseille, France, 2005.
- C. Bermúdez, R. M. Aguilar, and I. Castilla. Simulación de eventos discretos para la gestión de las listas de espera quirúrgicas y no quirúrgicas. In *Premios Profesor Barea a la Gestión y Evaluación de Costes Sanitarios*, pages 157–171. Fundación Signo, 2008.
- W. E. Biles and J. P. C. Kleijnen. International Collaborations in Web-based Simulation: A Focus on Experimental Design and Optimization. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 218–222. IEEE, 2005. ISBN 0-7803-9519-0. doi: 10.1109/WSC.2005.1574254. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1574254>.
- P. Bizarro, L. M. Silva, and J. a. G. Silva. JWarp: a Java library for parallel discrete-event simulations. *Concurrency: Practice and Experience*, 10(11-13):999–1005, 1998.
- C. J. M. Booth and D. I. Bruce. Stack-free process-oriented simulation. *ACM SIGSIM Simulation Digest*, 27(1):182–185, julio 1997.
- V. Bosilj-Vuksic, V. Ceric, and V. Hlupic. Criteria for the Evaluation of Business Process Simulation Tools. *Interdisciplinary Journal of Information, Knowledge, and Management*, 2:73–88, 2007. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.9869&rep=rep1&type=pdf>.
- L. Böszörményi and A. Stopper. Semi-automatic parallelization of object-oriented simulations. *Simulation Practice and Theory*, 7(4):295–307, junio

1999. ISSN 09284869. doi: 10.1016/S0928-4869(99)00014-2. URL <http://linkinghub.elsevier.com/retrieve/pii/S0928486999000142>.
- S. C. Brailsford, V. A. Lattimer, P. Tarnaras, and J. C. Turnbull. Emergency and on-demand health care: Modelling a large complex system. *The Journal of the Operational Research Society*, 55(1):pp. 34–42, 2004. ISSN 01605682. URL <http://www.jstor.org/stable/4101825>.
- R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977.
- A. Burns and G. Davies. *Concurrent programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1993. ISBN 0-201-54417-2.
- A. Buss. Component based simulation modeling with simkit. *Proceedings of the Winter Simulation Conference*, pages 243–249, 2002. doi: 10.1109/WSC.2002.1172891. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1172891>.
- Modsim II. The Language for Object-Oriented Programming. Reference Manual*. CACI Products Company, 1995.
- Y. Callero and R. M. Aguilar. Use of Simulation in eGovernment Process Development. A Case Study Using the Simulation Tool SIGHOS. In *21st European Modeling & Simulation Symposium proceedings*, Tenerife, Spain, 2009.
- R. S. d. Carvalho and J. G. Crookes. Cellular simulation. *Operational Research Quarterly (1970-1977)*, 27(1):31–40, 1976. ISSN 00303623. URL <http://www.jstor.org/stable/3009208>.
- R. A. Cassel. *Web-Based Simulation: The Three-Phase Worldview and Java*. PhD thesis, Lancaster University, Lancaster, 2000.
- R. A. Cassel and M. Pidd. Distributed discrete event simulation using the three-phase approach and Java. *Simulation*, 8:491–507, 2001.

- I. Castilla and R. M. Aguilar. Java for Parallel Discrete Event Simulation: a Survey. In R. M. Aguilar, A. G. Bruzzone, and M. A. Piera, editors, *21st European Modeling & Simulation Symposium proceedings (vol. I)*, pages 72–79, Puerto de la Cruz, Tenerife, Spain, 2009.
- I. Castilla, R. Muñoz, P. J. Baquero, and R. M. Aguilar. Helpdesk Modeling and Simulation with Discrete Event Systems and Fuzzy Logic. In *19th European Modeling & Simulation Symposium proceedings*, Genoa, Italy, 2007.
- I. Castilla, M. M. Günal, M. Pidd, and R. M. Aguilar. Hada: Towards a generic tool for data analysis for hospital simulations. In *European Modeling and Simulation Symposium EMSS 2008*, 2008.
- K. Chandy and R. Sherman. Space-Time and Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 53–57, Tampa, Florida, USA, 1989.
- K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, 1979. ISSN 0098-5589. doi: <http://dx.doi.org/10.1109/TSE.1979.230182>.
- A. C. Chow, B. P. Zeigler, and D. H. Kim. Abstract simulator for the parallel DEVS formalism. In *Fifth Annual Conference on AI, and Planning in High Autonomy Systems*, pages 157–163, Gainesville, FL, USA, 1994. IEEE Comput. Soc. Press. ISBN 0-8186-6440-1. doi: 10.1109/AIHAS.1994.390488. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=390488>.
- L. Chwif, M. R. P. Barretto, and R. J. Paul. On simulation model complexity. In J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *2000 Winter Simulation Conference Proceedings*, pages 449–455. IEEE, 2000. ISBN 0-7803-6579-8. doi: 10.1109/WSC.2000.899751. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=899751>.
- J. K. Cochran and A. Bharti. A multi-stage stochastic methodology for whole hospital bed planning under peak loading. *International Journal of Industrial*

- and Systems Engineering*, 1(1/2):8, 2006. ISSN 1748-5037. doi: 10.1504/IJISE.2006.009048. URL <http://www.inderscience.com/link.php?id=9048>.
- J. C. Comfort. The simulation of a master-slave event set processor. *Simulation*, 42(3):117–124, 1984. ISSN 0037-5497. doi: 10.1177/003754978404200304. URL <http://sim.sagepub.com/cgi/doi/10.1177/003754978404200304>.
- A. Concepcion. A hierarchical computer architecture for distributed simulation. *IEEE Transactions on Computers*, 38(2):311–319, 1989. ISSN 00189340. doi: 10.1109/12.16512. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=16512>.
- M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/366663.366704>.
- J. Cowie. Jted: parallel discrete-Event simulation in java. *Concurrency: Practice and Experience*, 10(11-13):993–997, septiembre 1998. ISSN 1040-3108. doi: 10.1002/(SICI)1096-9128(199809/11)10:11/13<993::AID-CPE405>3.0.CO;2-C. URL <http://doi.wiley.com/10.1002/%28SICI%291096-9128%28199809/11%2910%3A11/13%3C993%3A%3AAID-CPE405%3E3.0.CO%3B2-C>.
- O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/365813.365819>.
- F. Fatin. *A Programming Structure for Parallel Simulation*. PhD thesis, Brunel University, 1996.
- A. Ferscha and M. Richter. Massively parallel simulation of business process models. In *European Simulation Multiconference 1996*, pages 377–381, Budapest, Hungary, 1996.
- A. Ferscha and S. K. Tripathi. Parallel and Distributed Simulation of Discrete Event Systems. Technical report, University of Maryland at College Park, 1996.

- D. Fone, S. Hollinghurst, M. Temple, A. Round, N. Lester, A. Weightman, K. Roberts, E. Coyle, G. Bevan, and S. Palmer. Systematic review of the use and value of computer simulation modelling in population health and health care delivery. *Journal of Public Health Medicine*, 25(4):325–335, 2003. doi: 10.1093/pubmed/fdg075. URL <http://jpubhealth.oxfordjournals.org/cgi/content/abstract/25/4/325>.
- R. M. Fujimoto. Feature Article—Parallel Discrete Event Simulation: Will the Field Survive? *INFORMS JOURNAL ON COMPUTING*, 5(3):213–230, 1993. doi: 10.1287/ijoc.5.3.213. URL <http://joc.journal.informs.org/cgi/content/abstract/5/3/213>.
- R. M. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99. (Cat. No.PR00155)*, pages 46–53, Atlanta, GA, 1999. IEEE Comput. Soc. ISBN 0-7695-0155-9. doi: 10.1109/PADS.1999.766160. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=766160>.
- R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc, New York, NY, USA, 2000.
- R. M. Fujimoto and D. M. Nicol. Parallel simulation today. *Annals of Operations Research*, 53(1):249–285, diciembre 1994. ISSN 0254-5330. doi: 10.1007/BF02136831. URL <http://www.springerlink.com/index/10.1007/BF02136831>.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- Y. García-Hevia. Modelado y simulación en Internet. Master thesis, Escuela Técnica Superior de Ingeniería Informática. Universidad de La Laguna, 2008.
- J. M. Garrido and K. Im. Teaching object-oriented simulation with psimj simulation package. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional*

- conference, pages 422–427, New York, NY, USA, 2004. ACM. ISBN 1-58113-870-9. doi: <http://doi.acm.org/10.1145/986537.986643>.
- E. Glinsky and G. A. Wainer. New Parallel Simulation Techniques of DEVS and Cell-DEVS in CD++. In *39th Annual Simulation Symposium (ANSS'06)*, pages 244–251, Washington, DC, 2006. IEEE Comput. Soc. ISBN 0-7695-2559-8. doi: 10.1109/ANSS.2006.32. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1612865>.
- B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, Upper Saddle River, NJ, 2006.
- M. Günal and M. Pidd. Simulation Modelling for Performance Measurement in Healthcare. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 2663–2668. IEEE, 2005. ISBN 0-7803-9519-0. doi: 10.1109/WSC.2005.1574567. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1574567>.
- M. M. Günal. *Simulation Modelling for Understanding Performance in Healthcare*. Phd thesis, Lancaster University Management School, 2008.
- M. M. Günal and M. Pidd. Understanding Accident and Emergency Department Performance using Simulation. In *Proceedings of the 2006 Winter Simulation Conference*, pages 446–452. IEEE, Dec. 2006. ISBN 1-4244-0501-7. doi: 10.1109/WSC.2006.323114. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4117638>.
- M. M. Günal and M. Pidd. Interconnected DES models of emergency, outpatient, and inpatient departments of a hospital. In *2007 Winter Simulation Conference*, pages 1461–1466. IEEE, Dec. 2007. ISBN 978-1-4244-1305-8. doi: 10.1109/WSC.2007.4419757. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4419757>.

- M. M. Günal and M. Pidd. DGHPsim: Supporting smart thinking to improve hospital performance. In *2008 Winter Simulation Conference*, pages 1484–1489. IEEE, Dec. 2008. ISBN 978-1-4244-2707-9. doi: 10.1109/WSC.2008.4736228. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4736228>.
- B. A. W. V. Halderen and B. J. Overeinder. Fornax: Web-based distributed discrete event simulation in Java. *Concurrency: Practice and Experience*, 10(11-13):957–970, septiembre 1998. ISSN 1040-3108. doi: 10.1002/(SICI)1096-9128(199809/11)10:11/13<957::AID-CPE393>3.0.CO;2-4. URL <http://doi.wiley.com/10.1002/%28SICI%291096-9128%28199809/11%2910%3A11/13%3C957%3A%3AAID-CPE393%3E3.0.CO%3B2-4>.
- M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, and E. Bu. Maximizing multi-processor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996. ISSN 00189162. doi: 10.1109/2.546613. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=546613>.
- B. P. Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ, 1973.
- K. J. Healy and R. A. Kilgore. Silk: A Java-based Process Simulation Language. In S. Andrabttir, K. J. Healy, D. H. Withers, and B. L. Nelson, editors, *Proceedings of the 1997 Winter Simulation Conference*, pages 475–482. IEEE, 1997.
- P. Heidelberger. Discrete Event Simulations and Parallel Processing: Statistical Properties. *SIAM Journal on Scientific and Statistical Computing*, 9(6):1114–1132, 1988. ISSN 01965204. doi: 10.1137/0909077. URL <http://link.aip.org/link/SJOCE3/v9/i6/p1114/s1&Agg=doi>.
- P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In O. Balci, R. P. Sadowski, and R. E. Nance, editors, *Proceedings of the 22nd conference on Winter simulation*, pages 734–737, New Orleans, Louisiana, USA, 1990. IEEE Comput. Soc. Press.

- K. Helsgaun. Discrete Event Simulation in Java. Technical report, Roskilde University, 2004.
- J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.
- J. Himmelspach, R. Ewald, S. Leye, and A. M. Uhrmacher. Parallel and distributed simulation of Parallel DEVS models. In *Proceedings of the 2007 spring simulation multiconference*, volume 1, pages 249–256, Norfolk, Virginia, 2007. Society for Computer Simulation International.
- C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/355620.361161>.
- F. Howell and R. Mcnab. simjava: a discrete event simulation library for java. In *International Conference on Web-Based Modeling and Simulation*, pages 51–56, 1998. doi: 10.1.1.52.2940.
- M. Hybinette. Just-in-time cloning. In *18th Workshop on Parallel and Distributed Simulation, 2004. PADS 2004.*, pages 45–51. IEEE, 2004. ISBN 0-7695-2111-8. doi: 10.1109/PADS.2004.1301284. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1301284>.
- M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, 2001. ISSN 10493301. doi: 10.1145/508366.508370. URL <http://portal.acm.org/citation.cfm?doid=508366.508370>.
- P. Jacobs. *The DSOL simulation suite - Enabling multi-formalism simulation in a distributed context*. Phd thesis, TU Delft, 2005.
- P. Jacobs and A. Verbraeck. Single-Threaded specification of process-Interaction formalism in java. In *Proceedings of the 2004 Winter Simulation Conference*,

- pages 479–486. IEEE, 2004. ISBN 0-7803-8786-4. doi: 10.1109/WSC.2004.1371497. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1371497>.
- P. Jacobs, N. Lang, and A. Verbraeck. D-Sol; a distributed java based discrete event simulation architecture. In *Proceedings of the Winter Simulation Conference*, pages 793–800. IEEE, 2002. ISBN 0-7803-7614-5. doi: 10.1109/WSC.2002.1172962. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1172962>.
- I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1992.
- D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, julio 1985. ISSN 01640925. doi: 10.1145/3916.3988. URL <http://portal.acm.org/citation.cfm?doid=3916.3988>.
- D. W. Jones. *Concurrent simulation*. ACM Press, New York, New York, USA, 1986. ISBN 0911801111. doi: 10.1145/318242.318468. URL <http://portal.acm.org/citation.cfm?doid=318242.318468>.
- J. B. Jun, S. H. Jacobson, and J. R. Swisher. Application of discrete-event simulation in health care clinics: A survey. *Journal of the Operational Research Society*, 50(2):109–123, 1999.
- W. D. Kelton, R. P. Sadowski, and D. A. Sadowski. *Simulation with Arena*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 2002. ISBN 0071122397.
- G. Kesidis and A. Singh. An Overview of Cell-Level ATM Network Simulation. In *Proc. High Performance Computer Systems*, Montreal, PQ, 1995. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.1610>.
- B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. Phd thesis, Queensland University of Technology, 2002.

- B. Kiepuszewski, A. ter Hofstede, and W. van Der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, marzo 2003. ISSN 0001-5903. doi: 10.1007/s00236-002-0105-4. URL <http://www.springerlink.com/Index/10.1007/s00236-002-0105-4>.
- T. Kiesling. Progressive Time-Parallel Simulation. *20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*, pages 82–91, 2006. doi: 10.1109/PADS.2006.31. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1630712>.
- P. J. Koopman, Jr. *Stack computers: the new wave*. Halsted Press, New York, NY, USA, 1989. ISBN 0-470-21467-8.
- A. Kunert. Optimistic-parallel, process-oriented des in java using bytecode rewriting. Technical report, Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany, May 2008.
- L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. ISSN 00010782. doi: 10.1145/359545.359563. URL <http://portal.acm.org/citation.cfm?doid=359545.359563>.
- D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, Dec. 2005. ISSN 01676423. doi: 10.1016/j.scico.2005.03.007. URL <http://linkinghub.elsevier.com/retrieve/pii/S0167642305000663>.
- P. L'ecuyer and E. Buist. Simulation in java with ssj. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 611–620. IEEE, 2005. ISBN 0-7803-9519-0. doi: 10.1109/WSC.2005.1574301. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1574301>.
- Y.-B. Lin and E. D. Lazowska. A time-division algorithm for parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):73–83, 1991.

- ISSN 10493301. doi: 10.1145/102810.214307. URL <http://portal.acm.org/citation.cfm?doid=102810.214307>.
- M. L. Loper. *Approximate Time and Temporal Uncertainty in Parallel and Distributed Simulation*. PhD thesis, Georgia Institute of Technology, 2002.
- B. D. Lubachevsky. Bounded lag distributed discrete event simulation. In *SCS Multiconference on Distributed Simulation*, pages 183–191, 1988.
- E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software: Practice and Experience*, 26(3):327–356, 1996. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199603)26:3<327::AID-SPE12>3.0.CO;2-H. URL <http://doi.wiley.com/10.1002/%28SICI%291097-024X%28199603%2926%3A3%3C327%3A%3AAID-SPE12%3E3.0.CO%3B2-H>.
- R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 281–288, New York, NY, USA, 1992. ACM. ISBN 0-89791-542-9. doi: <http://doi.acm.org/10.1145/143457.143530>.
- R. B. v. d. Meer, L. A. Rymaszewski, H. Findlay, and J. Curran. Using or to support the development of an integrated musculo-skeletal service. *The Journal of the Operational Research Society*, 56(2):pp. 162–172, 2005. ISSN 01605682. URL <http://www.jstor.org/stable/4102167>.
- J. A. Miller, K. Kochut, W. D. Potter, E. Ucar, and A. Keskin. Query-driven simulation using active kdl: A functional object-oriented database system. *International Journal in Computer Simulation*, 1(1), 1991.
- J. A. Miller, R. Nair, Z. Zhang, and H. Zhao. JSIM: A Java-based simulation and animation environment. *Proceedings of 1997 SCS Simulation Multiconference*, pages 31–42, 1997. doi: 10.1109/SIMSYM.1997.586473. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=586473>.

- J. A. Miller, A. F. Seila, and X. Xiang. The JSIM web-based simulation environment. *Future Generation Computer Systems*, 17(2):119–133, octubre 2000. ISSN 0167739X. doi: 10.1016/S0167-739X(99)00108-9. URL <http://linkinghub.elsevier.com/retrieve/pii/S0167739X99001089>.
- L. Moreno, R. M. Aguilar, J. Piñeiro, J. Sigut, J. Estévez, C. A. Martín, and J. Sánchez. Patient-Centered Simulation to Aid Decision-Making in Hospital Management. *Simulation-Transactions of the Society for Modeling and Simulation International*, 74(5):290–303, 2000.
- L. Moreno, R. M. Aguilar, J. Piñeiro, J. Sigut, J. Estévez, and C. González. Using KADS methodology in a simulation assisted knowledge based system: application to hospital management. *Expert System with Applications*, 20:235–249, 2003.
- R. Muñoz. XMLGHOS: Una Interfaz para Especificar Modelos de Eventos Discretos Usando SIGHOS. Master thesis, Escuela Técnica Superior de Ingeniería Informática. Universidad de La Laguna, 2006.
- R. Muñoz and I. Castilla. SIGHOS sourceforge. Web site: <http://sourceforge.net/projects/sighos/> [Last accessed: 29 September 2010], 2010.
- N. Mustafee, K. Katsaliaki, and S. J. Taylor. Profiling Literature in Healthcare Simulation. *SIMULATION*, 86(8-9):543–558, 2010. doi: 10.1177/0037549709359090. URL <http://sim.sagepub.com/content/86/8-9/543.abstract>.
- S. Muthu, L. Whitman, and S. H. Cheraghi. Business process reengineering: A consolidated methodology. In *Proceedings of the 4 th Annual International Conference on Industrial Engineering Theory, Applications, and Practice, 1999 U.S. Department of the Interior - Enterprise Architecture*, pages 8–13, 2006.
- R. E. Nance. On Time Flow Mechanisms for Discrete System Simulation. *Management Science*, 18(1):59–73, 1971. URL <http://www.jstor.org/stable/2629293>.

- A. Neely, M. Gregory, and K. Platts. Performance measurement system design: A literature review and research agenda. *International Journal of Operations and Production Management*, 15(4):80–116, 1995.
- D. M. Nicol. Performance bounds on parallel self-initiating discrete-event simulations. *ACM Trans. Model. Comput. Simul.*, 1(1):24–50, 1991. ISSN 1049-3301. doi: <http://doi.acm.org/10.1145/102810.102812>.
- D. M. Nicol and P. F. Reynolds, Jr. Problem oriented protocol design. *ACM SIGSIM Simulation Digest*, 16(2):27–30, 1985. ISSN 01636103. doi: 10.1145/1102958.1102961. URL <http://portal.acm.org/citation.cfm?doid=1102958.1102961>.
- E. Niewiadomska-Szynkiewicz, M. Zmuda, and K. Malinowski. Application of a java-based framework to parallel simulation of large-scale systems. *Applied Mathematics and Computation*, 13(4):537–547, 2003.
- B. Page and W. Kreutzer. *The Java Simulation Handbook – Simulating discrete Event Systems with UML and Java*. Shaker Publishing, Aachen, Germany, 2005. ISBN 978-3832237714.
- H. Park and P. A. Fishwick. A fast hybrid time-synchronous/event approach to parallel discrete event simulation of queuing networks. *2008 Winter Simulation Conference*, pages 795–803, diciembre 2008. doi: 10.1109/WSC.2008.4736142. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4736142>.
- C. D. Pedgen, R. E. Shannon, and R. P. Sadowski. *Introduction to Simulation Using SIMAN, 2nd Ed.* McGraw-Hill, Inc., New York, 1995.
- K. Perumalla and R. Fujimoto. Efficient large-scale process-oriented parallel simulations. In D. Medeiros, E. Watson, J. Carson, and M. Manivannan, editors, *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*, pages 459–466. IEEE, 1998. ISBN 0-7803-5133-9. doi: 10.1109/WSC.1998.

745022. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=745022>.
- K. S. Perumalla. Parallel and distributed simulation: traditional techniques and recent advances. In *Proceedings of the 2006 Winter Simulation Conference*, pages 84–95. IEEE, diciembre 2006. ISBN 1-4244-0501-7. doi: 10.1109/WSC.2006.323041. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4117594>.
- P. Peschlow, A. Voss, and P. Martini. Good news for parallel wireless network simulations. In *Proceedings of the 12th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems - MSWiM '09*, pages 134–142, New York, New York, USA, 2009. ACM Press. ISBN 9781605586168. doi: 10.1145/1641804.1641828. URL <http://portal.acm.org/citation.cfm?doid=1641804.1641828>.
- M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, mayo 2000. ISSN 1040-3108.
- M. Pidd. Object-orientation, discrete simulation and the three-phase approach. *The Journal of the Operational Research Society*, 46(3):362–374, 1995. ISSN 01605682. URL <http://www.jstor.org/stable/2584330>.
- M. Pidd. *Computer Simulation in Management Science*. John Wiley & Sons, Inc., New York, NY, USA, 1998. ISBN 0471979317.
- P. F. Reynolds. Active Logical Processes and Distributed Simulation: An analysis. In S. Roberts, J. Banks, and B. Schmeiser, editors, *Proceedings of the 1983 Winter Simulation Conference*, pages 263–265, Arlington, Virginia, USA, 1983. IEEE Comput. Soc. Press.
- P. F. Reynolds, Jr. A shared resource algorithm for distributed simulation. In *Proceedings of the 9th annual symposium on Computer Architecture*, pages 259–266, Austin, Texas, USA, 1982. IEEE Comput. Soc. Press.

- A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0136744095.
- N. Russell, A. H. M. Hofstede, W. M. P. Van Der Aalst, and N. Mulyar. *Work-flow Control-Flow Patterns: A Revised View*. Technical report, BPMcenter.org, 2006.
- S. Sardesai, D. Mclaughlin, and P. Dasgupta. *Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs*. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 57–65, Las Vegas, Nevada, 1998.
- T. Schriber and D. Brunner. *Inside Discrete-Event Simulation Software: How it Works and Why it Matters*. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, editors, *Proceedings of the 2006 Winter Simulation Conference*, pages 118–128. IEEE, 2006. ISBN 1-4244-0501-7. doi: 10.1109/WSC.2006.323044. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4117597>.
- T. J. Schriber and D. T. Brunner. *Inside simulation software: how it works and why it matters*. In *Proceedings of the 1994 Winter Simulation Conference*, pages 45–54, Washington, DC, USA, 1994. IEEE Computer Society.
- L. Schruben. *Simulation modeling with event graphs*. *Commun. ACM*, 26(11): 957–963, 1983.
- M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262692155.
- SNS. *Indicadores Clave del Sistema Nacional de Salud*. Ministerio de Sanidad y Consumo, March 2007. URL <http://www.msc.es/estadEstudios/estadisticas/sisInfSanSNS/pdf/indicadoresClaveCISNS.pdf>.

- A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. ISBN 9780136006633.
- R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/360303.360308>.
- Y. M. Teo and Y. K. Ng. Spades/java: object-Oriented parallel discrete-Event simulation. *Proceedings 35th Annual Simulation Symposium. SS 2002*, pages 245–252, 2002. doi: 10.1109/SIMSYM.2002.1000160. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1000160>.
- The Apache Software Foundation. JavaFlow homepage. Web site: <http://commons.apache.org/sandbox/javaflow/> [Last accessed: 29 September 2010], 2010.
- The Jikes Team. Jikes homepage. Web site: <http://jikes.sourceforge.net/> [Last accessed: 29 September 2010], 2010.
- The NHS Information Centre. HES Online Data. Web site: <http://www.hesonline.org.uk> [Last accessed: 29 September 2010], 2008.
- The RIFE team. RIFE homepage. Web site: <http://www.rifers.org/> [Last accessed: 29 September 2010], 2010.
- W. M. P. Van Der Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. doi: 10.1023/A:1022883727209.
- R. M. Weatherly and E. H. Page. Efficient process interaction simulation in java: implementing co-Routines within a single java thread. In *Proceedings of the 2004 Winter Simulation Conference*, pages 373–379. IEEE, 2004. ISBN 0-7803-8786-4. doi: 10.1109/WSC.2004.1371483. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1371483>.

- WfMC. Workflow Management Coalition Terminology & Glossary. Technical Report WfMC-TC-1011, Workflow Management Coalition, Winchester, Hampshire, UK, February 1999. URL <http://www.wfmc.org/Download-document/WfMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html>.
- T. Wiedemann. Discrete Event Simulation with Universal Programming Languages on Multicore Processors. In *Proceedings of the 20th European Modeling and Simulation Symposium - EMSS 2008*, Campora San Giovanni, Amantea, Italy, 2008.
- Workflow Patterns Initiative. WorkFlow Patterns homepage. Web site: <http://www.workflowpatterns.com/> [Last accessed: 29 September 2010], 2010.
- M. T. Wynn, M. Dumas, C. J. Fidge, A. H. M. Hofstede, and W. M. P. V. D. Aalst. Business Process Simulation for Operational Decision Support. In A. Hofstede, B. Benatallah, and H.-Y. Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 66–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78237-7. doi: 10.1007/978-3-540-78238-4_8. URL <http://www.springerlink.com/index/10.1007/978-3-540-78238-4>.
- B. Zarei. Parallel Simulation for Business Process Re-engineering. In M. Paprzycki, L. Tarricone, and L. T. Yang, editors, *Practical parallel computing*, pages 163–183. Nova Science Publishers, Inc, 2001.
- B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modelling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000.