

THE DSOL SIMULATION SUITE

ENABLING MULTI-FORMALISM SIMULATION IN A
DISTRIBUTED CONTEXT

THE DSOL SIMULATION SUITE

ENABLING MULTI-FORMALISM SIMULATION IN A DISTRIBUTED CONTEXT

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op dinsdag 15 november 2005 om 10.30 uur

door

Peter Hubertus Maria JACOBS

bestuurskundig ingenieur

geboren te Utrecht.

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. H.G. Sol

Prof. dr. ir. A. Verbraeck

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter

Prof. dr. H.G. Sol, Technische Universiteit Delft, promotor

Prof. dr. ir. A. Verbraeck, University of Maryland, USA, promotor

Prof. dr. R.W. Wagenaar, Technische Universiteit Delft

Prof. dr. S. Boyson, University of Maryland, USA

Prof. dr. R.E. Nance, Virginia Tech, USA

Prof. dr. ir. J.J.M. Evers, Technische Universiteit Delft

Prof. dr. ir. J.C. Wortmann, Rijksuniversiteit Groningen

Download this thesis from <http://www.peter-jacobs.com/documents/thesis.pdf>

Download DSOL from <http://www.simulation.tudelft.nl>

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, Den Haag

Jacobs, Peter Hubertus Maria

The DSOL simulation suite/

Peter Hubertus Maria Jacobs - [S.l. : s.n.].

Proefschrift Delft. - Met Index, lit. opg., Nederlandse samenvatting

ISBN-10: 90-9019835-0

ISBN-13: 978-90-9019835-4

NUR 992

Trefw.: decision support, simulation, Java, object-orientation.

Cover design: Bas Uildriks

Printing: Copie Sjob, Delft, the Netherlands - <http://www.copie-sjob.nl>

English editor: Miranda Aldham-Breary M.Sc. P.G.C.E.

Copyright ©2005 by P.H.M. Jacobs

All rights reserved worldwide. No part of this thesis may be copied or sold without the written permission of the author.

to my family.

The applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The author does not offer any warranties or representations, nor does he accept any liabilities with respect to them.

Contents

1. EFFECTIVE DECISION SUPPORT	1
1.1 Introduction	1
1.2 Human decision making and problem solving	2
1.3 Simulation as a method of inquiry	3
1.4 Effectiveness of decision support systems	5
1.5 A generation of systems supporting substantive rationality	5
1.6 A simulation suite to support bounded rationality	6
1.6.1 Studio based decision making	6
1.6.2 Service oriented computing	7
1.6.3 Research question	8
1.7 Research approach	9
1.7.1 Philosophy	9
1.7.2 Strategy	10
1.7.3 Instruments	11
1.8 Research outline	13
2. SIMULATION IN PRACTICE	15
2.1 Case 1: The net-centric supply chain	15
2.1.1 Introduction	15
2.1.2 Relevance	17
2.1.3 Conceptualization	18
2.1.4 Specification	19
2.1.5 Conclusions	23
2.2 Case 2: Controlling automated guided vehicles	26
2.2.1 Introduction	26
2.2.2 Relevance	27
2.2.3 Conceptualization	27
2.2.4 Specification	29
2.2.5 Conclusions	32
2.3 Research question revised	33

3. SYSTEMS ENGINEERING PRINCIPLES	35
3.1 Systems engineering	35
3.2 Principles for system design	37
3.3 Object-oriented system description	38
3.4 Principles for object-oriented modeling	40
3.5 Summary	46
4. SIMULATION AS A METHOD OF INQUIRY	47
4.1 Actors and activities in a simulation study	47
4.2 A framework for simulation	48
4.3 Multi-formalism modeling	49
4.4 Three classical formalisms for discrete event simulation	53
4.5 Experimental design	55
4.6 Activities involved in a simulation study	56
4.7 Requirements for a simulation suite	57
4.8 Requirements worked out	58
4.8.1 Usefulness	59
4.8.2 Usability	60
4.8.3 Usage	61
4.9 Summary	62
5. DESIGNING A SIMULATION SUITE	63
5.1 Distribution forms the core of DSOL	63
5.2 Choosing an object-oriented programming language	64
5.3 An overview of Java based simulation environments	65
5.4 Overview of the DSOL simulation suite	66
5.5 Specification of distributed asynchronous communication	70
5.6 Specification of formalisms	74
5.6.1 Discrete event formalism in detail	80
5.6.2 Process interaction formalism in detail	81
5.6.3 Differential equation formalism in detail	87
5.7 Specification of statistical distribution functions	88
5.7.1 Pseudo random number generators in detail	88
5.7.2 Statistical distributions in detail	91
5.8 Specification of output statistics	93
5.9 Specification of animation	95
5.10 Summary	96

6. VERIFICATION AND TESTING OF DSOL	99
6.1 Expert verification through the SNE comparisons	99
6.1.1 Comparison 1: lithium-cluster dynamics	101
6.1.2 Comparison 2: flexible assembly system	106
6.1.3 Comparison 3: generalized class-E amplifier	111
6.1.4 Comparison 6: emergency department	116
6.1.5 General conclusions on the SNE comparisons	124
6.2 Testing and analyzing the DSOL suite	126
6.2.1 Code formatting and style checking	126
6.2.2 Unit testing	128
6.2.3 Profiling	128
6.3 Conclusions	130
7. CASE: EMULATION WITH DSOL	133
7.1 Introduction	133
7.1.1 60m ³ of concrete floors on an automated guided vehicle	134
7.1.2 The importance of emulation in the design of realtime control	136
7.1.3 Expectations for the case	136
7.1.4 Requirement analysis	137
7.2 Conceptualization	138
7.2.1 The conceptual model of the control system	139
7.2.2 The conceptual model of the controlled system	140
7.2.3 The model-PLC interface	140
7.3 Specification	144
7.3.1 Modbus communication with DSOL	144
7.3.2 The specification of emulation components	144
7.3.3 The specification of the DSOL-PLC communication	148
7.3.4 Experimentation	149
7.4 Conclusions	149
8. CASE: FLIGHT SCHEDULING AT KLM	153
8.1 Introduction	153
8.1.1 The plan acceptance process	154
8.1.2 Why a renewed specification in DSOL?	156
8.2 Conceptualization	157
8.3 Specification	161
8.3.1 Input specification	161
8.3.2 Output specification	164
8.4 Conclusions	165

9. EPILOGUE	169
9.1 A review of research questions	169
9.2 The DSOL user community	172
9.3 Recommendations for further research	173
Appendix	175
A Time advancing functions of simulators	177
B The specification of the port example	181
C Java Naming and Directory Interface	184
D DSOL experiment file	185
E JUnit test of DSOL's discrete event list	186
Index	199
Author index	200
Subject index	203

ACKNOWLEDGMENTS

For the realization of this thesis I am indebted to many people. In the first place I want to thank Henk Sol for never being satisfied, for his inspiring advice, and for reminding me, again and again, that I should concentrate on my dissertation. A second word of acknowledgment must be directed to Alexander Verbraeck for inspiring me to start this Ph.D. research, for being an equally enthusiastic Java programmer, and for inspiring me *not* to only concentrate on my dissertation. Both promotors formed an excellent and pleasant team to work for and with.

I would like to thank Niels Lang and Stijn-Pieter van Houten for all the discussions, and above all, for their valuable contributions to the DSOL suite; many thanks!

I am greatfull to the US Department of Defense, Dycore, Frog Navigation Systems, KLM Royal Dutch Airlines and TBA Nederland b.v. for their willingness to provide the real life environments in which parts of this research were conducted. I truly hope that the work presented in this thesis helps them in their challenging decision making situations.

I would like to thank Miranda Aldham-Breardy for improving my English. I would like to thank Wieke Bockstael-Blok, Roy Chin and Tamrat Teweldeberhan for reading the thesis and providing me with valueable feedback.

I want to thank Rutger Heeren and Matthijs Visser for providing me with beers even when I knew it would be better to be asleep. I want to thank Bas Uildriks, Cinco Veldman and Louis Veldman for designing and printing this beautiful thesis. Most of all I want to thank my family for all their love and support and Marieke for interrupting her journey and attending the defense.

Peter H.M. Jacobs
Delft, November 2005

1. EFFECTIVE DECISION SUPPORT

1.1 Introduction

In a recent survey by Barr (2003) it was reported that, while 66 percent of managers believe decision support systems are critical to analyze operational business processes, only 40 percent actually uses them. In this thesis we will argue that a new paradigm is needed for the design of decision support systems to support unstructured decisions effectively and thus to support human decision making with information technology. This is extra important in a digitally connected world in which decision makers are continuously overloaded with an unrestricted amount of information, and where support tools are available any place, any time.

In their book *Rehearsing the future*, Keen and Sol (2005) present an overview of decision support, viewing it as a lens, an invitation and a field of practice. The focus of this lens is to enhance executive decision making using a new generation of decision support technology. Keen and Sol argue that decision support has two practical goals. One, to support technology professionals, consultants and specialist to leverage their contribution to decision-makers through effective design and use of models, information systems and tools. Two, to support executives in public, private and public-private organizational contexts to improve the *effectiveness* of their decision process as it builds their confidence in, and their comfort with using, appropriate models, systems and tools.

As will become apparent throughout the remainder of this chapter, we underwrite the goal expressed in *Rehearsing the future* that the focus of next generation decision support should be on the support of procedural, or bounded rationality to improve the effectiveness and efficiency of problem solving in a multiple actor, multiple view environment.

Human decision making forms the central theme of this chapter. First the distinction between procedural and substantive decision making is considered. Then simulation is introduced as the method of inquiry to be supported. A framework for expressing the effectiveness of decision support tools is presented in section 1.4. The problem statement for the research is discussed in section 1.5 after which the research question is presented in section 1.6. We conclude this chapter with our research strategy and thesis outline.

The focus of next generation decision support should be...

...on the support of procedural, or bounded rationality, which is...

1.2 Human decision making and problem solving

More than a century ago, Cournot made some preliminary remarks on the role of mathematics when applied to the administrative sciences. Cournot (1838) saw that where a market is supplied by only a few producers, the notion of profit-maximization is ill-defined. The rational choice for each actor depends on the choices made by the other actors; no actor can choose without making assumptions about how other actors will make their choices.

Although this notion was published in 1838, Simon (1976) states that it was only after the second world war that classical economic theory was supplemented by a perspective on economics that was based on *procedural, or bounded, rationality*. Classical economic theory rests on two fundamental assumptions. One, the economic actor has a well-defined particular goal. Two, the economic actor is *substantively rational*, which by definition stipulates that the rationality of the behavior of the actor depends on one aspect only: his or her goal.

Procedural rationality assumes that the concept of rationality is synonymous with *the peculiar thinking process called reasoning* (James, 1890). According to Simon, behavior is *procedurally rational* when it is the outcome of appropriate deliberation. Accepting *procedural rational* behavior therefore makes the process of problem solving, or decision making, not a theory of best solutions, of *substantive rationality*, but a theory of efficient activities, i.e. to find good, or accepted, solutions (Simon, 1976). Decisions become a result of *satisficing* instead of *optimization*.

...the outcome of appropriate deliberation.

In addition to the ill-defined goals observed by Cournot (1838), Simon (1955) describes several other factors that diminish our *substantive rationality* and cause Man's decisions to be based on *procedural rational behavior*.

- Man's computational efficiency: no processing equipment exists, for most problems faced by human decision makers, that will enable them to discover the substantive optimal solution, even when optimal is well defined.
- Substantive rational decision makers do not take the view of reaching a *satisficing level*. A typical example occurs when a decision maker receives a sequence of offers, and must decide whether to accept the highest offer before the next one is received. In such a case a decision maker reaches a level of *acceptance* or *satisficing*.
- Another reason for *procedural rationality* is the cost of information gathering. Simon argues that whenever this process incurs costs, the optimum is likely to be affected by the process.
- The theories on substantive rational decision making enforce a scalar goal, or value function. Simon points out that this value function might be a vector

function in the case of group decision with a pay-off function consisting of the individual values or in cases where values do not have a common *denominator*¹.

The challenge now becomes to support the human realities of decision making, or problem solving, by supporting N decision makers with M perceptions, P constraints and Q goals. In this support we distinguish the process, or method, from tools. We will introduce simulation as the method of inquiry to be supported with appropriate decision support tools, i.e. simulation tools.

The challenge becomes to support the human realities of decision making by supporting N decision makers with M perceptions, P constraints and Q goals.

1.3 Simulation as a method of inquiry

A systems view of problem solving is shown in figure 1.1 and the four elements which form the different stages of this process are delineated. The arrows in figure 1.1 are used to emphasize the different activities.

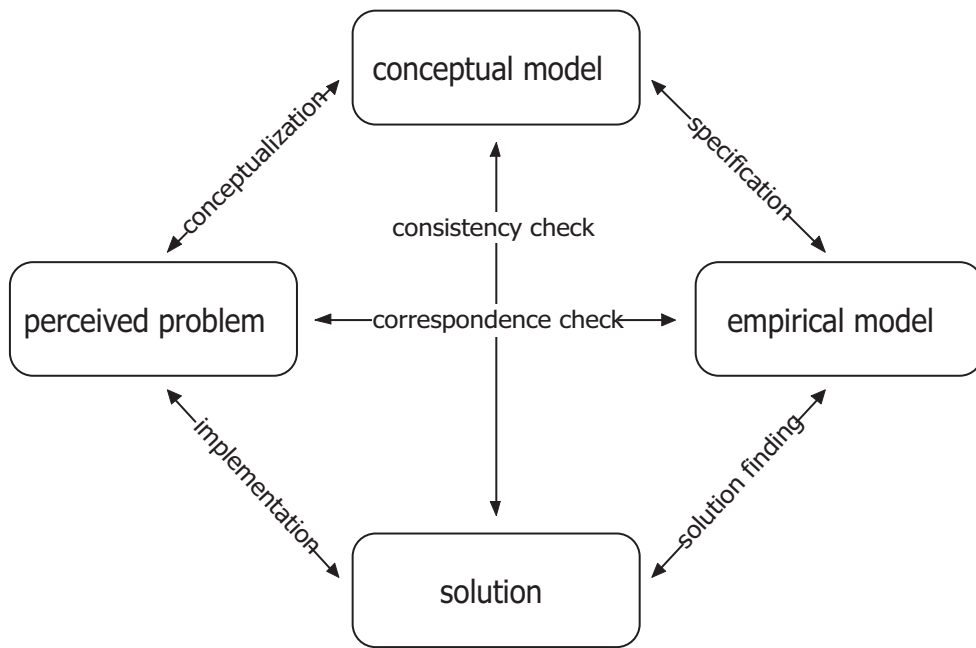


Fig. 1.1: A systems view of problem solving (Mitroff et al., 1974)

The four stages of the process of problem solving are the *perceived problem*, the *conceptual model*, in which the variables, that will be used to specify the nature

¹ Alternatives are valued based upon more than one attribute, e.g. price and age.

A specific combination of activities leads to a process of problem solving.

of the problem in broad terms, are defined, the *empirical model* in which the conceptual model in terms of the system under study is specified, and the solution. The activities illustrated in figure 1.1 are *conceptualization*, *specification*, *solution finding*, *implementation*, *consistency check* and *correspondence check*. A specific combination of these activities leads to a *model cycle*, i.e. a process of problem solving.

Perhaps the most important, and yet in many ways the most subtle fact presented in this figure is that, while all model cycles have a beginning, not all beginnings are the same (Nutt, 2002; Mitroff et al., 1974).

The set of activities that make up a specific process of problem solving can be supported with a structured set of instruments called an *inquiry system* (Sol, 1982). This concept is an extension of the *inquiring system* as defined by Churchman (1971); where Churchman focuses on activities and underlying philosophies, Sol includes the instruments to support these activities. Churchman (1971) presents five philosophers and describes how each philosopher would design inquiring systems to understand the relations in systems under their consideration. Following Churchman, the inquiring systems are listed below.

Churchman clearly advocates a *Singerian* inquiring system for ill structured problem solving.

- *Leibnitzian* in which a purely formal, deductive philosophy is reflected.
- *Lockean* in which an experiential, inductive and empirical philosophy is reflected.
- *Kantian* in which both formal and empirical philosophies are reflected. The Kantian inquiring system reconciles the Leibnitzian and Lockean inquiring systems.
- *Hegelian* in which a synthetic philosophy is reflected. Using a Hegelian inquiring system researchers aim to invoke the strongest possible conflict on any issue.
- *Singerian* in which a synthetic, interdisciplinary philosophy is reflected.

Sol presents simulation as a *Singerian* inquiry system.

Churchman (1971) clearly advocates a *Singerian* inquiring system for ill-structured problem solving. Sol (1982) presents simulation as a *Singerian* inquiry system and advocates that simulation is the preferred method of inquiry for ill-structured problems. In the remainder of this thesis, simulation is defined as the process of designing a model of a real system and conducting experiments with this model for the purpose of either understanding the behavior of the system or of evaluating various strategies for its operation (Shannon, 1975).

1.4 Effectiveness of decision support systems

A leading question in the development of decision support systems concerns the effectiveness of these systems. Keen and Sol (2005) argue that the effectiveness can be expressed in a combination of three Us: *usefulness*, *usability* and *usage*.

Effectiveness of DSS is usefulness,...

- *Usefulness*. The usefulness of decision support tools expresses the value they add to the decision making process. It thus relates to the analytic models, embedded knowledge and information resource available in a model or tool. Usefulness is commonly acquired through domain specific libraries, examples, templates and support.
- *Usability*. Usability expresses the mesh between people, processes and technologies. Usability depends mainly on the interface between users and the decision support technology. Usability expresses, among others, the responsiveness, flexibility, adaptability and ease of interaction and collaboration with the system.
- *Usage*. Usage expresses the flexibility, adaptivity and suitability of DSSs for organizational, technical, or social context. The main question concerning usage is: How is the system embedded in the decision process?

...usability

...and usage.

According to Keen and Sol, traditional decision support tools do not place equal emphasis on the three Us. While some provide extensive usefulness through detailed embedded knowledge, they lack usability due to severe platform constraints, i.e. they are written for one particular operating system, or they do not provide clear interfaces to external information resources: substantive rationality underlies their design.

1.5 A generation of systems supporting substantive rationality

Based on the evaluations of commercially available simulation environments by Hlupic (1993); Tewoldeberhan et al. (2002), we conclude that in most of the commercially available simulation environments only one actor can use a simulation model, designed by one simulation model designer, to carry out one experiment. Most simulation environments only support one model formalism, one reporting format and one framework for animation. The simulation environment commonly supports one operating system, one hardware platform, one processor and thus one concurrent physical location. Simulation environments are commonly designed for one type of user, and one domain with one set of key performance indicators.

Where a simulation environment should support a synthetic, interdisciplinary approach to problem solving,...

...the current generation of simulation environments supports a more substantive rational approach to solution finding.

Where a simulation environment should support a synthetic, interdisciplinary approach to problem solving, the current generation of simulation environments supports a more substantive rational approach to solution finding. We argue that most of the currently used simulation environments are based on a *1-1-1* paradigm while decision makers clearly require a converse $N_n-N_m-N_o$ paradigm for their decision support services and processes.

1.6 A simulation suite to support bounded rationality

The research presented in this thesis is founded on the hypothesis that the web-enabled, service oriented characteristics of information system development provide a clear opportunity to design a simulation suite that is based on a $N_n-N_m-N_o$ paradigm. With such suite a studio based decision process is to be supported.

1.6.1 Studio based decision making

Stressing all three Us equally results in a DSS studio, which is a (virtual) environment in which...

Stressing all three Us equally results in the concept of decision support *studios*, *suites* and *services* (Keen and Sol, 2005). A suite is a well chosen set of services and recipes for inter-connectivity; a decision support suite is thus a chosen set of services and recipes to support a decision making process. A studio is a (virtual) environment in which suites are deployed, e.g. a group decision room or a web-portal. Studio based decision support is identified by the following three elements.

- *Target*: the target for the domain of decision support should be the arena of ill-structured, multidisciplinary and multi-actor problems. This includes infrastructural expansions and supply chain coordination issues.
- *Process*: decision support should be embedded in a studio-based process. A studio is defined as *a facilitative environment, face-to-face or via telecommunication links, that enhances the active inclusion in the process of stakeholders and builds the collaboration that is an intrinsic requirement for effective processes* (Keen and Sol, 2005).
- *Technology*: decision support should include suites to support the studios. Suites are integrated IT development tools, systems and analytic methods that are explicitly aimed at enhancing the studio decision process.

...suites, i.e. sets of services and recipes for inter- connectivity, are deployed.

The introduction of a studio for decision support places the emphasis on leveraging the agility and effectiveness of decision support systems to a level that has already been achieved in more operational management based decision making. An attractive example of the innovate path we aim to follow is illustrated in figure 1.2. A decision making studio is presented in this figure: here executive decision makers are supported by a suite of interacting models and information resources.



Fig. 1.2: Strategic management cockpit (SAP, 2004)

1.6.2 Service oriented computing

We argue that the concept of a suite is explicitly related to the *service oriented society* we live in. In a service oriented society, craftsmen are, due to specialization and standardization, replaced by *service providers*, i.e. organizations that implement a service, supply its description and provide related technical and business support (Papazoglou and Dubray, 2004).

The *service oriented computing (SOC)* paradigm, or *service oriented architecture (SOA)*, that underlies the design of modern information systems is presented in figure 1.3. Papazoglou and Geogakopoulos (2003) define service oriented computing as the computing paradigm that utilizes services as fundamental elements for developing information systems; it is about bringing connectivity at the center of systems and software engineering. In this paradigm, a service is a self-describing, open component that supports the rapid, low-cost composition of a distributed information system. Service descriptions are used to advertise the service capabilities, interface, behavior and quality. Since services may be offered by different autonomous service providers, service oriented computing results in a distributed computing infrastructure for both intra- and inter-organizational information systems.

Although the service oriented computing paradigm has resulted in the better connectivity of distributed information services, it has also questioned all our traditional beliefs on the structure of information systems: software applications are

The concept of a suite is entwined with...

...service oriented computing, which is based on...

...services which are self-describing, open components.

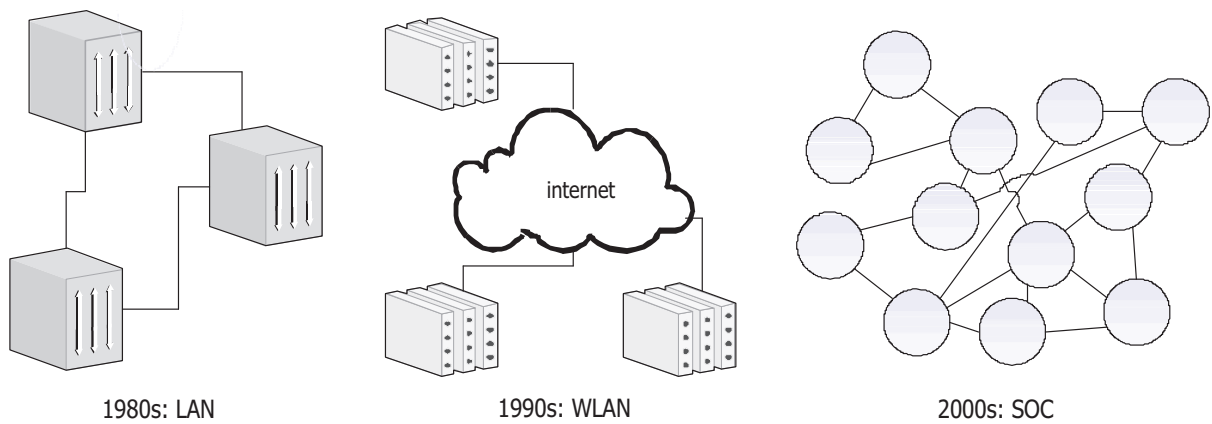


Fig. 1.3: The advent of the service oriented architecture

no longer single systems running on a single computer and bounded by a single organization, and while concurrency has become the norm for the deployment of service oriented information systems, programming languages and technical infrastructures are not designed for this (Papazoglou and Geogakopoulos, 2003).

The application of service oriented computing on the web is manifested in the concept of a *web service*. A web service is a specific kind of service that is identified by a uniform resource identifier (URI), the service description and transport components of which utilize open internet standards, e.g. the simple object access protocol (SOAP) and the web service description language (WSDL) (Papazoglou and Geogakopoulos, 2003).

Two final remarks must be made with respect to the context and value of a service. One, a service is mostly designed to neglect the context it is deployed in; this intended independence results in a *loosely coupled structure* between services. Two, the value of a service may well be created by combining, i.e. composing, services. The functions of such a *composite service* include the coordination of dataflow and the monitoring of the quality of the composed services.

1.6.3 Research question

We now introduce the research question addressed in this thesis, which is based on theories of studio based decision making and service oriented software engineering.

Research question: Can we create a simulation suite for decision makers that supports a studio-based decision process and improves their effectiveness when solving ill-structured, multidisciplinary problems?

A service is mostly designed to be independent of the specific context it is deployed in; this intended independence results in a *loosely coupled structure* between services.

Can we create an simulation suite for multiple decision makers, multiple models in multiple formalisms?

1.7 Research approach

A scientific inquiry may best be illustrated as following a particular *process* or *strategy* in which a set of *research instruments* are employed and which is guided by the researchers using an underlying *research philosophy*. In the following sections we discuss the *philosophy*, *strategy* and *instruments* applied in the pursuit of the research objectives presented in this thesis.

We follow a scientific process, which is guided by...

1.7.1 Philosophy

The ambitious question that one aims to answer with a research philosophy is: *How can I come to know something new about the universe?* Where some will rely on sensory impressions, others will rely on pure reason, and yet others on a scientific method. Determining the assumptions made in each approach allows us to choose better our own approach to the acquisition of knowledge.

...a research philosophy, e.g. activism or passivism.

One can distinguish two major *schools of thought* in the field of research philosophies (Wilson and Keil, 1999): *passivism* and *activism*. Passivists, or justificationists, believe that sensory experiences are impressed on a passive mind in a mechanistic way (Acton, 2004), they thus believe that knowledge has foundations. Descartes, with his *indubitable* clear and distinct ideas, and Hume, with his *incorrigible* sensory experiences, are typical examples. Recent passivists include the *logical positivists*, or *logical empiricists*, such as Carnap and Simon, who believe that observation statements are the foundation of all meaningful concepts (Hintikka, 1975). In the field of decision support, logical positivists have repeatedly aimed at the development of *a general problem solver* in which humans are fully substituted by computers (Newell and Simon, 1963; Simon, 1977; Klir, 1985). Although all passivists accept the objectivity of measurements and experiences, they follow different approaches with regard to scientific inquiry.

- *Inductivists* believe that science progresses through the accumulation of facts.
- *Probabilists* believe that repeated empirical verifications of a theory, i.e. successful predictions of observational results, make a theory more probably true.
- *Dogmatic, or naive, falsificationists* believe that science progresses by the generation of hypotheses which are falsified by nature.

Activists, also referred to as *conventionalists* or *interpretivists*, believe that sensory observations are always influenced by one's preconceptions, i.e. they are *theory-laden*. Thus, the mind actively constructs one's experiences of the environment, and descriptions of events are always *inferential*, *interpretive* and *theoretical*.

Activists thus deny that sensory observations are a sure foundation for knowledge; it is selective (Koningsveld, 1987). Scientific knowledge therefore becomes a human creation. Two different categories of activism can be distinguished.

- *Conservative activists*, referred to as Kantians. Kant believed that observations are structured by the *a priori* categories of the mind. The real world beyond our innate conceptual framework is, according to Kantians, either forever unknown or created by God to mirror the real world.
- Revolutionary conventionalists deny that science will ever become proven knowledge; all knowledge is fallible. All assumptions are subject to criticism and voluntary change. Revolutionary conventionalists are either realists such as Popper and Lakatos or anti-realists such as Duhem.

We postulate that systems engineering is a subjective human creation, and as such base our research on realistic activism.

Although philosophers from both *schools of thought* have formed an underlying philosophy for organizational and information system research, we postulate that systems engineering is a subjective human creation and as such base our research on realistic activism, or revolutionary conventionalism.

1.7.2 Strategy

The research presented in this thesis reflects the design of a simulation suite; that is the design of an information system. We present in this section a *research strategy* for the design of the suite, and thus present a strategy for the accomplishment of our research objective.

March and Smith (1995) present two strategies for the design of an information system: the behavioral-science paradigm and the design-science paradigm.

March and Smith (1995) present two strategies, or paradigms, for the design of an information system: the *behavioral-science* paradigm and the *design-science* paradigm. The behavioral-science paradigm has its roots in natural science research methods; it seeks to develop and justify theories, i.e. principles and laws, that explain or predict organizational and human phenomena surrounding the analysis, design, implementation, management and use of information systems (Delone and McLean, 1992, 2003; Seddon, 1997). The design-science paradigm has its roots in engineering and the sciences of the artificial (Simon, 1996). It is fundamentally a problem-solving paradigm. It seeks to create innovations that define the ideas, practices, technical capabilities and products through which the analysis, design, implementation and use of information systems can be effectively and efficiently accomplished (Tsichritzis, 1997; Denning, 1997).

Lee (2000) argues that technology and behavior are not dichotomous in an information system: they are inseparable. Hence, the need to address the pragmatists deriving from Denvey who basically argue that the validity of theory is the action it enables. Hevner et al. (2004) argue that technology and behavior are similarly inseparable in information system *research*. Philosophically this argument is draw

from Aboulafia (1991) who argues that truth, i.e. justified theory, and utility, i.e. systems that are effective, are two sides of the same coin and that scientific research should be evaluated in light of its practical implications. Hevner et al. (2004) argue based on the above that design-science and behavioral-science should be engaged in a complementary research paradigm. We agree with the arguments of Hevner et al. (2004) and postulate such complementary, explorative research strategy for this research. In this explorative strategy behavioral-science addresses research through the development and justification of theories that explain or predict phenomena related to the identified need. Design-science addresses research through the building and evaluation of artifacts designed to meet the identified business need.

The design-science paradigm is fundamentally a problem-solving paradigm.

Although a behavioral-science strategy is initially applied to understand organizational needs with respect to decision support, issues concerning the strategy, alignment and organizational design of decision support are outside the scope of this thesis. We thus mainly concentrate on the design-science strategy in the pursue of our research objective.

We mainly concentrate on the design-science paradigm in the pursue of our research objective.

To achieve a true understanding of and appreciation for design science, an important dichotomy must be faced. Design is both a process, i.e. a set of activities, and a product, or artifact; design is thus both a verb and a noun (Walls et al., 1992). March and Smith (1995) identify two design processes and four design products in information system research. The two processes are *build* and *evaluate*. The products are *constructs*, *models*, *methods* and *instantiations*. Constructs provide a language in which a problem is defined and communicated (Hevner et al., 2004). Models use constructs to represent the design problem and its solution space Simon (1996). Methods define the processes that provide guidance on how to search the solution space. Instantiations are implementations in a working system and provide a proof of concept of the solution.

1.7.3 Instruments

We discuss the selected research instruments to implement the research strategy in this section. The most commonly used instruments in the field of information system development are (Galliers, 1992):

- *laboratory experiment*: the investigation of relations between controlled variables, with minimal and tightly controlled variations, solving an artificial problem situation.
- *field experiment*: the experimentation of a small number of uncontrolled variables in a practical, i.e. existing, problem.
- *case study*: a planned and focused study of a phenomenon in its natural

setting with a large number of variables. Due to the natural setting, experimental control over the number and the variation of variables is limited or non-existent.

- *action research*: a study of relationships in the real world where the researcher is actively involved and has an influence on the outcome of the study. Action research can thus be described as a research methodology in which the researcher simultaneously pursues *action*, or change, and *research*, or understanding (Dick, 1999).
- *survey*: an investigation of a situation at a particular point in time, where there is an actual basis for collecting and assessing data from multiple cases. Surveys are commonly based on questionnaires which can be analyzed statistically.
- *theorem proof*: a rigorous mathematical argument which unequivocally demonstrates the truth of a given proposition within a given set of axioms and assumptions. A proven mathematical statement is called a *theorem* (Aigner and Ziegler, 1998). Hypotheses are validated by the construction of theorems based on a set of already validated derivation rules.
- *simulation*: the process of designing a model of a real system and conducting experiments with this model for the purpose of either understanding the behavior of the system or of evaluating various strategies for its operation (Shannon, 1975).
- *forecasting*: the processes used to make forecasts. Typically, forecasting is used to predict over time, i.e. time-series forecasting, and to make predictions about differences among people, firms, or other objects, i.e. cross-sectional data. The field includes the study and application of judgment and quantitative, statistical methods.

We place a strong emphasize on *neutrality* and *external validity* when selecting the research instruments for...

Action research, field experiments, laboratory experiments and case studies form logical instruments for the inductive exploration of a problem and an evaluation of design. The disadvantages of these instruments are that researchers may leap to conclusions based on limited data or that they may drop disconfirming evidence (Eisenhardt, 1989).

Given these disadvantages, and to limit their effects in our research, we place a strong emphasize on *neutrality* and *external validity* when selecting the research instruments to be used for the research presented here. Thus with respect to neutrality we chose to work on problems in which we had no vested interest, i.e. no stake in the outcome, and as a result, we chose not to use action research. *Expert validation* and *generalization* of findings through multiple cases were chosen as the

means to meet our aim of external validity of results. The question how we acquire such validity is discussed in the chapters presenting the specific cases.

1.8 Research outline

The outline of this research, illustrated in figure 1.4, reflects the explorative research strategy. Two explorative case studies are presented in chapter 2. In this chapter hypotheses are conclusively refined to complete the initial phase of this research.

The concepts and theories of *object-oriented* system design are introduced in chapter 3, which is concluded with a set of principles for object-oriented systems design. In terms of the design-science strategy, the object-oriented theories form constructs; the principles form the methods.

We introduce several theories of *modeling* and *simulation* in chapter 4: the history of simulation (Nance, 1995), *state-time relations* (Nance, 1981), a framework for simulation (Zeigler et al., 2000) and relations between simulation formalisms (Vangheluwe and de Lara, 2002). This chapter contains the model.

The actual design, or instantiation, is presented in chapter 5. Here we present our contribution to the field of systems engineering with the introduction of a distributed simulation object library (DSOL), and we discuss the requirements, architecture and implementation of the DSOL suite.

Verification and expert validation of DSOL are presented in chapter 6. In chapters 7 and 8 we present a validation of our hypotheses that DSOL contributes to more effective decision support, based on the 3 Us. We present our conclusions and explore potential future research in chapter 9.

...the explorative research strategy underlying this research.

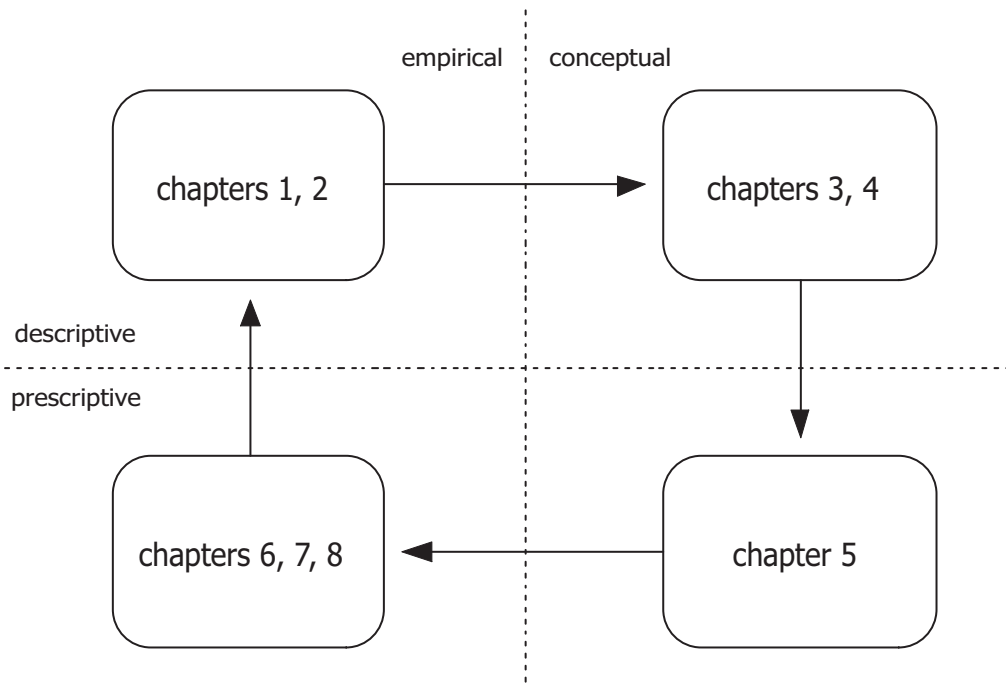


Fig. 1.4: The outline of this research

2. SIMULATION IN PRACTICE

We present two case studies in this chapter. The aim behind presenting these two real-life situations is to sharpen our understanding of what is required, with respect to usefulness, usability and usage, of a simulation environment. Tailored simulation models were developed in both case studies and as such both cases served as explorative basis for the requirements and design of the simulation suite discussed in chapter 5.

In this chapter we present two case studies.

We aim to provide insight into the effectiveness of a simulation environment based on 3 different roles one can distinguish in a simulation study: the role of the decision maker, the role of the simulation model builder and the role of systems engineer, i.e. the developer of the simulation environment. Although a more detailed introduction to the actors and roles involved in a simulation study is presented in section 4.1, we will link our explorative findings to the effectiveness of decision making per role per activity to introduce the framework for requirement analysis.

2.1 Case 1: The net-centric supply chain

2.1.1 Introduction

In 2001 Delft University of Technology joined a consortium¹ established to design a next generation infrastructure for superior command and control at the supply chain center of the United States Air Force (USAF) (Eccleston, 2002).

The assumption of the consortium was that an integrated information technology architecture could be designed to overcome the problems that plague many supply chains. Electronic exchange of information leads to a reduction in errors and increased efficiency of the working processes. When one company can use the information of other companies in the supply chain, the negative effects of uncertainty can, in theory, be mitigated. In practice, however, the exchange of

In the first case study we describe the design of a next generation infrastructure for superior command and control at the supply chain center of the United States Air Force.

¹ Among the governmental partners of the consortium were the Defense Supply Center at Richmond, the Air Combat Command Logistics Group at Langley, and the 7th Logistics Group at Dyess. Among the educational and industrial partners were the RH Smith Business School, Delft University of Technology, ASD, General Electric, Avaya, Oracle, Sun Microsystems, Manugistics, and Tibco/TMS.

information between companies is not as easy as it seems. Many different systems and standards are used, the number of peer-to-peer relations with other companies in the network is usually too large to manage, most systems are not open for easy exchange of information with other systems, and most companies are very reluctant to share information with other companies in the first place (Boyson et al., 2003).

The Department of Defense understood the importance of a new infrastructure to support real-time supply chain decision-making. Toward that end, the office of the Secretary of Defense sponsored a pilot project that would demonstrate the characteristics and effectiveness of a portal-based architecture for managing supply chains in the defense domain.

Portal technology allows all the partners in a supply chain to log onto a single site and immediately get the relevant information they need to make certain decisions.

Portal technology allows all the partners in a supply chain to log onto a single portal site and immediately get the relevant information they need to make certain decisions. The portal has uses for both suppliers and customers. Suppliers can be given insight into the inventory levels of other portal users and tune their products based on this information. Customers can be given diverse information and services via a front-end on the Internet. For instance, a customer can log onto the portal, enter an assigned security password, and gain access to real-time information about his or her order. The customer can check on the production or shipping status of an order, inventory availability or a host of other customer-specific data. This resulted in the following requirements for the consortium (Boyson et al., 2003):

- to enable radical simplification of supply chain user interface and working processes
- to demonstrate the end-to-end supply chain
- to provide real-time integration of, and visibility over, existing air force transactional systems
- to provide forward visibility over global processes
- to create virtual, integrated and real-time supply chains

The consortium built an alpha-version supply chain e-portal for the F101 engine community of the USAF and General Electric.

The consortium built an alpha-version supply chain e-portal for the F101 engine community of the USAF and General Electric, which provides web-based business functionality, asset visibility, and total system intelligence to both the Air Force and General Electric supply chain managers. The example supply chain of the low-pressure turbine for F101 engines involved twenty-five parts, all made by General Electric. Using this supply chain as a reference model, a comprehensive electronic platform was built that combined field data collection technology, ERP functions, advanced planning, collaborative planning and forecasting and real-time control

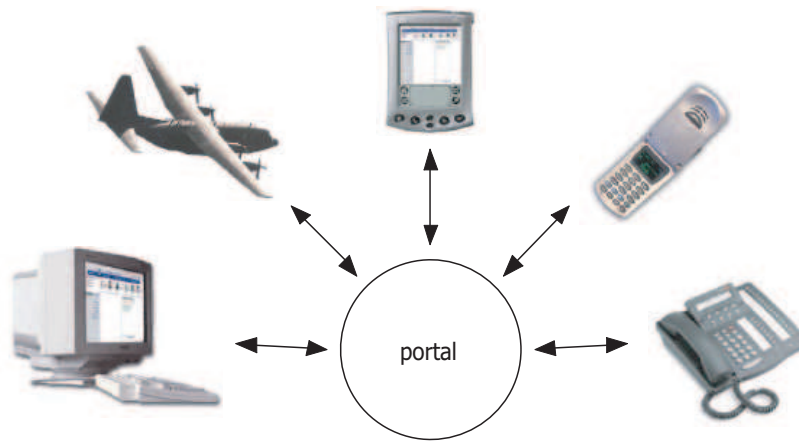


Fig. 2.1: Multi-channel portal (Eccleston, 2002)

panel displays using geographic visualization and arrays of key performance indicators. A password-protected test portal was built for evaluation by participants in the supply chain to allow experimentation with these kinds of online functionality (Boyson et al., 2003).

2.1.2 Relevance

We address the relevance of this particular case for our research in this section. What makes this case so attractive for our research? To answer this, we recall the $N_n-N_m-N_o$ paradigm of chapter 1, and see that in this particular case the following N s are explicitly required:

- multiple geographically and organizationally dispersed decision makers to experiment concurrently within the portalled decision support studio
- multiple client side operating systems and multiple hardware platforms to access the portal
- multiple composite models, fed by distributed data sources, to forecast the performance of the supply chain

We furthermore argue that the autonomous services provided by the different actors in the F101 supply chain form a strong background for the service oriented computing paradigm presented in section 1.6.2 on page 7. Our overall conclusion was that this case lent itself well to help us to understand to what extent traditional simulation environments support the required portalled support studio, and to determine what services are required to fulfill the goals of the consortium.

A detailed requirement analysis disclosed an unbridgeable gap between the requirements of the case and the features of commercially available, i.e. traditional, simulation environments.

2.1.3 Conceptualization

We, a team of researchers from the Systems Engineering group of Delft University, provided the simulation and visualization services for the supply chain portal. The most relevant aspect of the case was to actually support multiple decision makers. Generals, procurement officers, commercial traders and others all are supposed to see, understand and experiment with portalled simulation models to address the issues in supply chain logistics.

A detailed requirement analysis disclosed an unbridgeable gap between the requirements formulated by the consortium and the features of traditional simulation environments which are based on the concept of substantive rational decision making (Boyson et al., 2003).

- Traditional simulation environments do not separate a model from a simulator. Such a separation is required to replay the deterministic past, in which case the decision support environment is fed with actual data and thus no simulator is used to generate representative data.
- Traditional simulation environments do not separate a modeling environment from an experimentation environment, i.e. output and control screens. Such separation is required to implement a multi-channel architecture where the model is executed on one or more back-end systems and users are asynchronously subscribed to outputs specifically tailored for their device.
- Security is a key requirement for any military development project (Eccleston, 2002). Security resulted in a requirement that simulation models may not be executed on a client device. Clients should merely receive updated animations of results. How could traditional simulation be applied to fulfill such constraints?
- Most traditional simulation environments are platform dependent (Hlupic, 1993; Tewoldeberhan et al., 2002) and as such undeployable on the Java/UNIX-based servers proposed by the other partners in the consortium. Traditional environments are furthermore single threaded and do not scale on multiple-CPU environments. How were these traditional simulation environments ever going to handle the gigabytes of transactions of the distributed supply chain in real time?

To meet the above requirements and constraints, a choice was to develop of a Java based simulation portal. Among the most important reasons to use this programming language were its platform independency, its wide-spreading usage among the partners involved and its support for distributed, web-enabled programming. The architecture of the case is presented in figure 2.2. The first tier is the *por-*

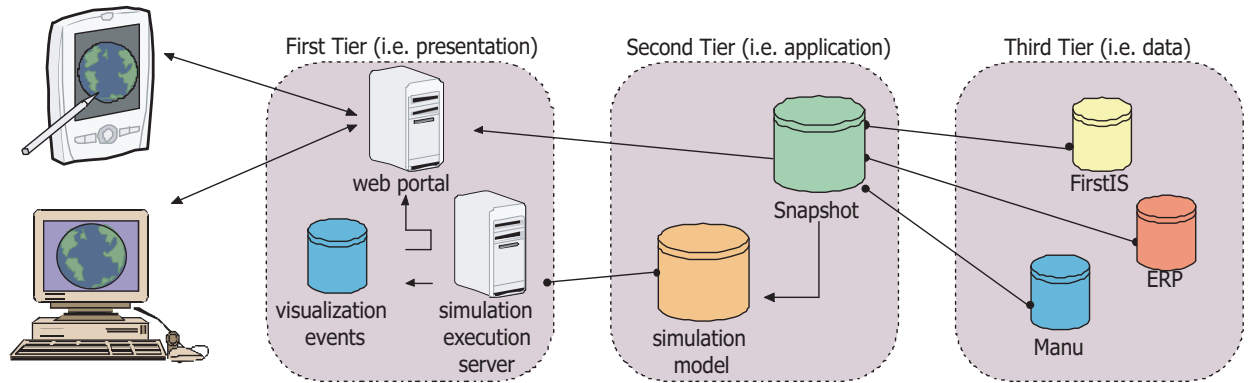


Fig. 2.2: Architecture of the DoD simulation portal

tal tier which provides a web-based, single point of access to users. The second tier, i.e. the *application and middleware tier*, provides the Java based simulation model. The third tier is the *database tier* consisting of enterprise resource planning, decision support, warehouse and supply chain management systems of the F101 engine. *Distributed computing* is a key concept for the specification of this architecture; all of the back-end systems were geographically distributed over US Air Force locations. As a result, all services, including the proposed simulation services, needed to support this distributed interaction. The Air Force required *multi-channeling* within the first tier. Access had to be based on the availability of devices and tools (see figure 2.1).

A firewall separates the presentation tier, i.e. the client or the first tier, from the other tiers in figure 2.2. The first tier, i.e. the client tier, receives visualization events which are to be presented in the portal. The dedicated simulation server instantiates simulation models from a model factory and initial values are fetched from underlying data sources.

This gap led to the development of a Java-based simulation environment.

2.1.4 Specification

The Java programming language was used to implement the architecture presented figure 2.2, because it is a platform independent, object oriented programming language, and because it was the de-facto standard of all the other partners in the consortium.

The specification of the case can be divided into three parts reflecting the three tiers in the conceptual model. These parts are the specification of the portal tier, the specification of the application layer tier and the specification of the data tier.



Fig. 2.3: Specification of the data tier

Specification of the data tier

The Java programming language provides JDBC technology as an application program interface to provide cross-database connectivity to a wide range of SQL databases. JDBC technology was used in this case study to parse the actors, i.e. the suppliers and the buyers, in the F-101 supply chain. A class diagram of the specification of the data tier is presented in figure 2.3. The `DoDObjectFactory` class is used to parse customers and suppliers using the `java.sql.Connection` interface. Although we explored the need to make business rules, actors, and policies more flexible, no proven concepts or object libraries were available. As a result, we introduced tailored supply chain classes such as the `DoDTrader`, the `Base`, etc.

Specification of the application tier

The simulation model was specified in the application tier, as illustrated in the class diagram in figure 2.4.

The simulation model was specified in the application tier, as illustrated in the class diagram in figure 2.4. We see in this class diagram that the `SimModelInterface` does not contain the model, i.e. a representation of the F-101 supply chain, but the state and behavior of a simulator, e.g. the `start` and `pause` functionality.

We furthermore see that the `SimModelInterface` contains an `EventList` on which `SimEvents` are to be scheduled and removed. A `SimEvent` contains a target on which a method must be invoked at a specified simulation time. In this situation the model is thus only implicitly available as it forms the content of the `eventList`.

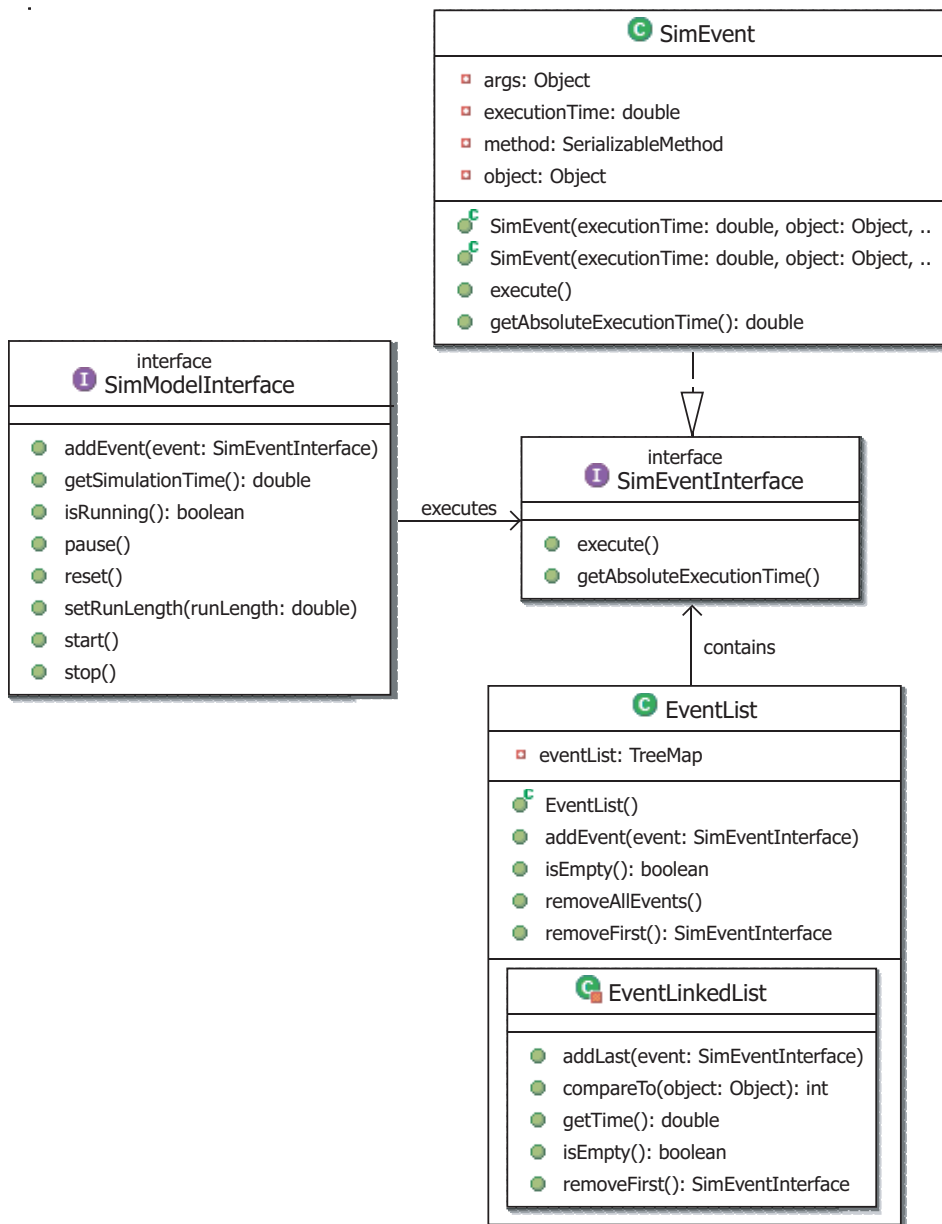


Fig. 2.4: Specification of the simulation model

A more in-depth discussion on the nature of a `SimEvent` is presented in section 5.6.1 on page 80.

Specification of the portal tier

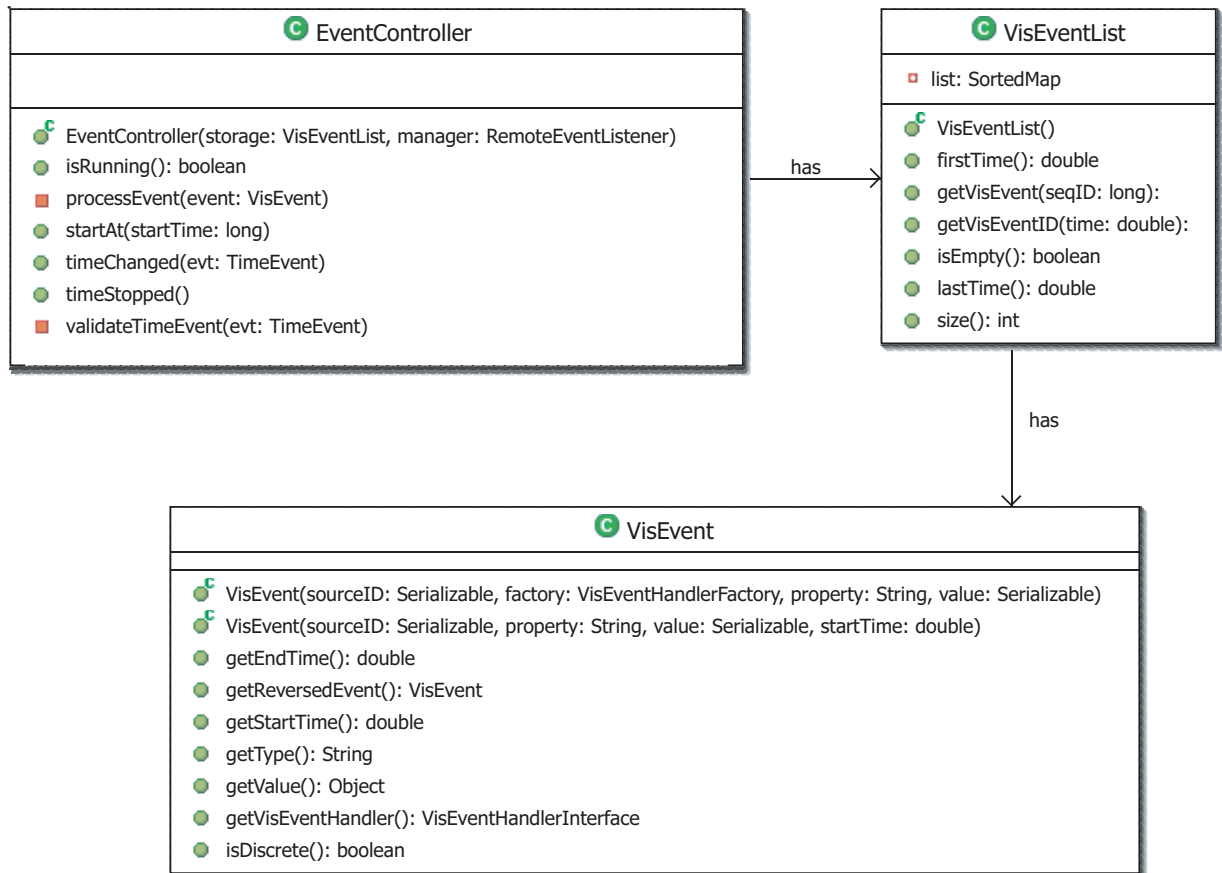


Fig. 2.5: Specification of client side visualization

The specification of the portal tier is illustrated in figure 2.5. The class diagram illustrated in this figure is deployed in a client-side applet. The `EventController` functions as a client-side animator which can be stopped, resumed and started. We see that `VisEvents` are fired from model objects to the portal tier where they are stored in a `VisEventList`. A `VisEvent` contains information for the applet on how to visualize a state change.

The specification of the portal tier fulfills three requirements. One, visualizations implement the requirement for distributed animation since they are fired by model objects in the application tier to applets in the presentation tier; in other words

they are fired from a server side simulation model to a client side web-page. Two, although visualization events represent server side model objects, they do not contain the business rules of the model objects. This ensured that neither model logic nor model behavior were available to the client. Three visualization events can represent both simulated data and the actual deterministic past.

To limit the number of `VisEvents` transported from the application tier to the portal tier, the visualization events are based on the principle of *dead reckoning*, i.e. the process of estimating a position by advancing a known position using course, speed, time and distance to be traveled (Cai et al., 1999). Since dead reckoning algorithms had to be specified for each individual model object, the relationship between model objects, e.g. aircrafts, and the client side animation panels was considered to be too complex.

To limit the number of `VisEvents` transported from the application tier to the portal tier, the visualization events were based on the principle of dead reckoning.

2.1.5 Conclusions

The outcome of the case is presented in figure 2.6: a client side, portalled presentation and control of simulation models. We successfully specified a distributed model, distributed visualization and a deterministic replay of the past while conforming to the required security constraints. This distributed visualization and control of the simulation model contributed mainly to the useability of the simulation environment for the decision makers; the generals were given web-based access to models representing their supply chain.

We successfully specified a portalled deployment of simulation models, but...

Based on Boyson et al. (2003) we can argue that the success of this explorative case study was thus directly related to supporting $n > 1$ decision makers. In a distributed world where different people are depending on each other while taking decisions, we clearly validated both the need and our ability to support $n > 1$ decision makers.

Further success, with respect to fulfilling the requirements presented on page 17, that was based on user experience and assesment, formed a clear justification for further research into the development of a simulation suite (Jacobs et al., 2002). A number of conclusions regarding further research were drawn.

- We conclude that since every single object, e.g. airplane, order, part, could physically be deployed on a different computer, the concept of an identified model with clear system boundaries evaporated. The increased usefulness for model builders, achieved by providing facilities to use distributed model components, clearly affected the usability of the simulation environment.
- Model objects fire visualization events to asynchronously subscribed animation panels. We concluded that the required dead-reckoning made client side animation too complex. A useable structure for animation and visualization remained a topic for further research.

...system boundaries evaporated, i.e. the architecture was too flexible,...

...animation was too complex ...

...and the model was
hard coded.

- In the field of software engineering the term *hard coded* refers to unchangeable code. Hard-coded features are built into the software in such a way that they cannot be easily modified. A final conclusion in this case study was that the model was hard coded due to the unavailability of a standard supply chain library; decision makers were not able to change the policies that represented their behavior, e.g. ordering, billing and transportation policies. This we consider an issue for both the decision maker and the model builder with respect to usefulness and usability. Not providing a framework for experimentation makes the environment less useable and therefore less useful with respect to decision making.

A more general conclusion is expressed by Verbraeck (2004):

”Without doubt, supply chain simulation is a technology to watch. Recent advances are transforming this technology from one that presented a static, cumbersome and often outdated view of a supply chain to something entirely different. New simulation technology, combined with the power of internet-based, distributed, real-time processing capabilities, is giving birth to an entirely new generation of modeling and simulation tools. These tools will present managers with an up-to-the-minute view of the activities of their extended global supply chains. Web-based simulations, real-time interfaces between simulation models and data sources, and distributed simulations will give managers the power to make real-time decisions in a real-time world.”

Verbraeck (2004) clearly presents the challenges and opportunities in supply chain modeling: supply chain managers need to be well served by web-based, real-time simulation possibilities. The product of this case formed a first step towards meeting this challenge.

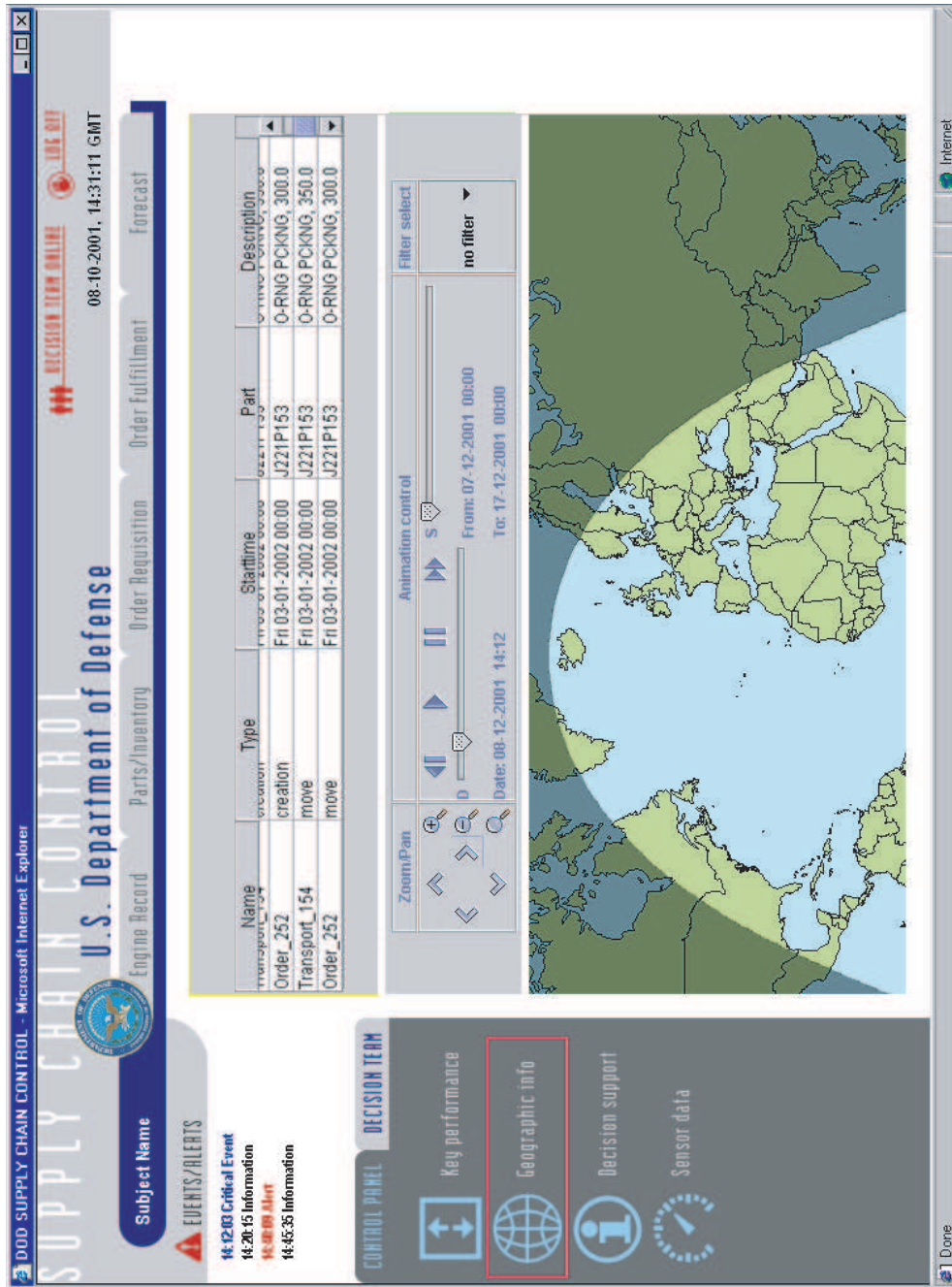


Fig. 2.6: US Air Force net-centric supply chain portal (Boyson et al., 2003)



Fig. 2.7: Park-A-Car (Frog, 2004)

2.2 Case 2: Controlling automated guided vehicles

2.2.1 Introduction

In 2002 Frog navigation joined a tender from the city of Rotterdam for the design of a sophisticated parking garage. The municipality planned, because of space constraints, a 2 floor underground parking garage in which cars would be parked by automatic guided vehicles. Frog navigation proposed a solution named *Park-A-Car* which is illustrated in figure 2.7.

The second case study involved the design of a 2 floor underground parking garage in which cars would be parked by automatic guided vehicles.

Park-A-Car is an automated parking system consisting of dual mode *side loaders* efficiently racking and stacking cars in a warehouse environment. Cars are placed bumper-to-bumper, which decreases required building volume by up to 40 percent. A specially developed side loader, the parking shuttle, picks up a car by its tires using two sets of telescopic forks; it then enters a warehouse environment using narrow aisles and unloads the loaded car onto one of the *shelves* on either side. The actual number of parking shuttles used depends on the capacity of the parking garage and the required retrieval speed for cars from the warehouse space (Frog, 2004).

Because of capacity constraints and the use of sophisticated technologies, the municipality of Rotterdam required a simulation model to be developed that would

reassure it that no motorist would have to wait for more than 15 minutes for his or her car. This simulation study was conducted by Delft University and served as explorative case for our research into the development of a simulation suite.

2.2.2 Relevance

To illustrate the value of this particular case for our research, we recall the N_n - N_m - N_o paradigm of chapter 1. We see that the following N s were explicitly required in this particular case:

- multiple distributed stakeholders had to experiment concurrently within the simulation environment. These decision makers included Frog's engineering team, Frog's sales team and the municipality of Rotterdam.
- multiple hardware and software platforms had to be supported by the simulation model, e.g. Linux, Microsoft Windows.
- the conceptual model of the case was based on multiple formalisms, i.e. world-views. The movement of the automatic guided vehicles was conceptualized as a set of differential equations, and passenger arrival was conceptualized in a discrete event formalism.

The value of this case mainly resulted from the requirement that multiple distributed simulation model builders had to be supported.

2.2.3 Conceptualization

A CAD drawing of the first floor (level -2) of the parking garage is presented in figure 2.8. In the center of the drawing a number of crosses indicate the elevators by which cars enter and leave the garage. On both sides of these elevators conveyors function as a buffer and as pickup location for the shuttles.

We will first outline the questions submitted by the Municipality of Rotterdam and Frog's engineering team before we elaborate on the challenges of this particular case study for our research.

- What is the minimum, average and maximum motorist waiting time?
- How many automatic guided vehicles, i.e. shuttles, are needed, and what is their utilization?
- What is the optimal elevator-vehicle-lane allocation scheme?
- What route and semaphore allocation model should be applied to minimize interference between shuttles and prevent potential collision?
- What filling scheme should be used to deal with arrival peeks in the early morning and departure peeks around 6pm?

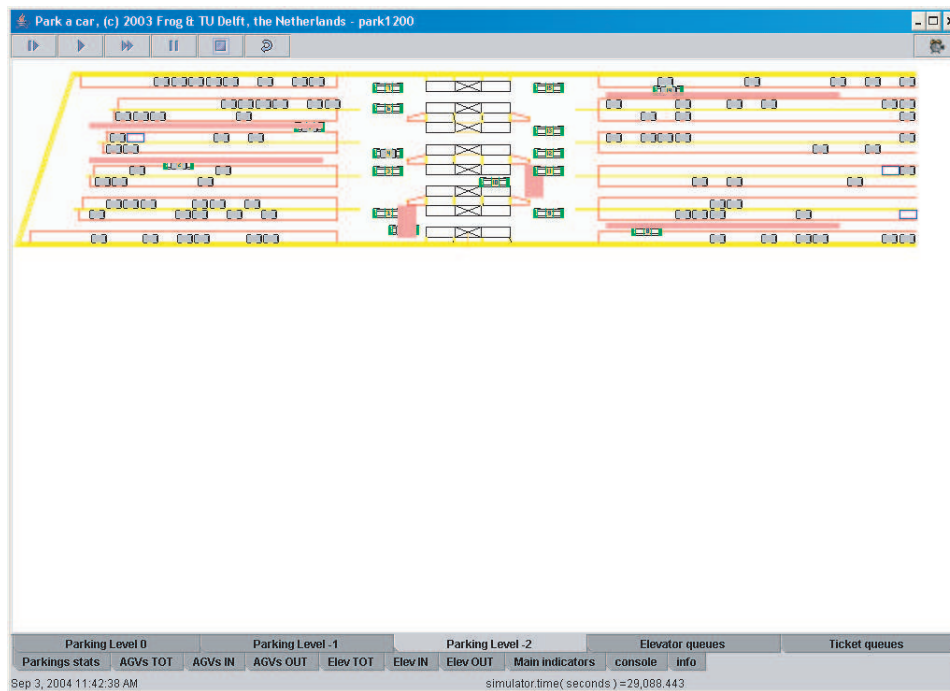


Fig. 2.8: Animation of the simulation model

The first activity in the simulation approach is to conceptualize the problem into a set of models representing the structure and the processes of the case. A partial class diagram, conceptualizing the structure of the case, is presented in figure 2.10. In this figure the **Parking** class specifies the parking garage containing elevators, ticket machines, etc. The interface based scheme selection, i.e. the **FillschemeInterface**, illustrates the extent to which Frog’s engineering team was able to test different fill and release scenarios.

By far the most interesting class of figure 2.10 is the **AGV** class. As its name suggest this class represents the automatic guided vehicles, i.e. the side loaders, of the case. From a conceptual point of view, the behavior of an **AGV** would best be described as a sequence of actions. An example of such sequence is presented in figure 2.9. Here we illustrate the loading process of an **AGV**. An **AGV** drives, whenever requested by an elevator, to the elevator, potentially unloads an already loaded vehicle and loads a new vehicle. Such a conceptualization of the behavior of an object in its *process* method reflects a modeling paradigm known as *process interaction*. A more in depth discussion on process interaction is presented in section 4.4 on page 53 and section 5.6.2 on page 81.

From a conceptual point of view, the behavior of an **AGV** could be described as a sequence of actions.

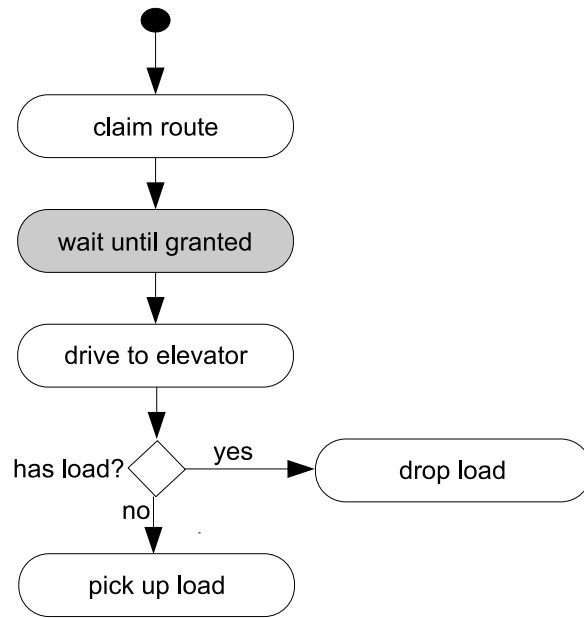


Fig. 2.9: Loading process of an AGV

2.2.4 Specification

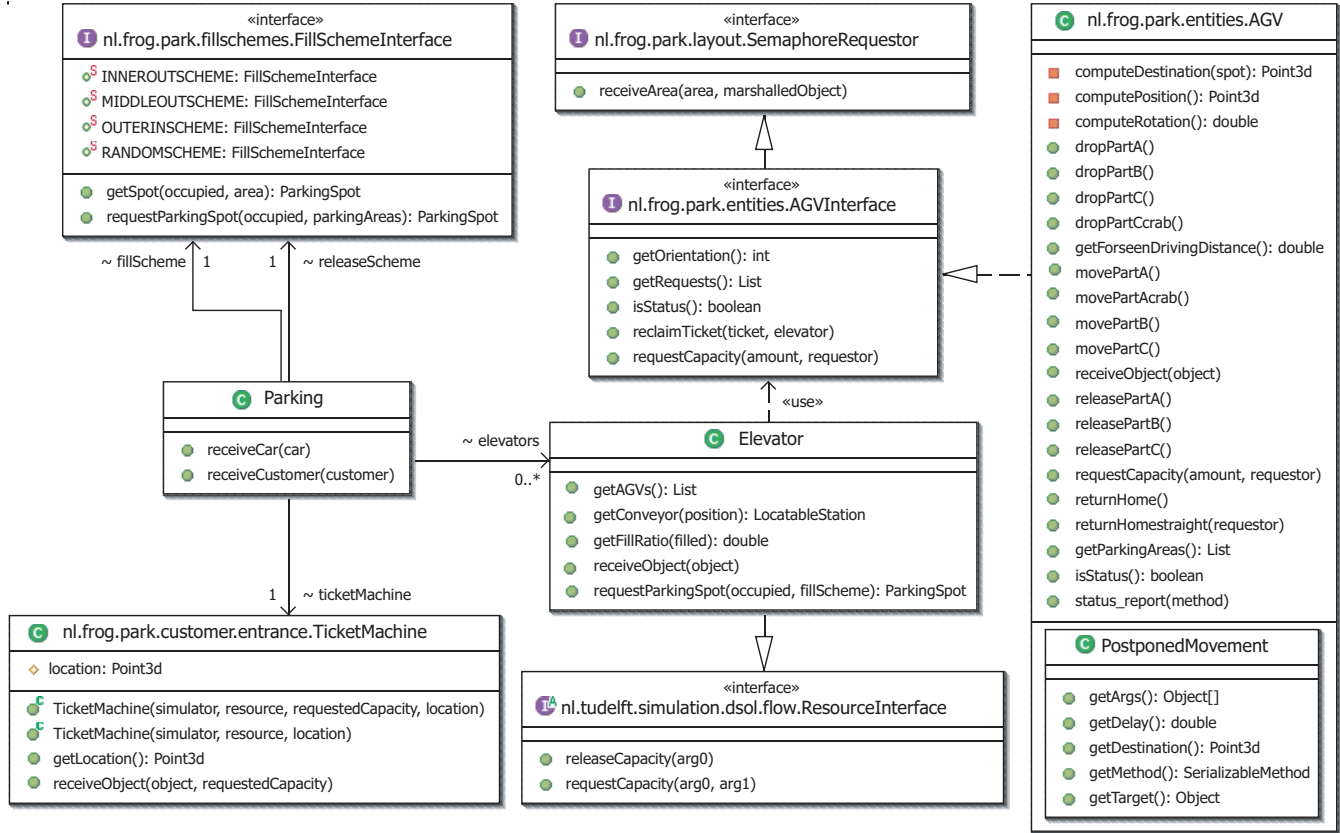
Although process interaction was supported in early simulation languages such as *Simula 67* (Birtwistle, 1979), current object oriented programming languages poorly support it. The reason is that process interaction requires an ability to suspend and resume a process; this is illustrated by the *wait-until-granted* block in figure 2.9. A more detailed discussion of this problem is presented in sections 5.3 and 5.6.2.

Since our Java based simulator did not support the process interaction formalism, we were forced to specify the simulation model in another formalism: the discrete event formalism. The conceptual model and its Java specification thus could not be specified in the same formalism. As a consequence we had to split the process into small subprocesses which we could schedule on a discrete event list. The large number of almost equally named methods in the *AGV* class in figure 2.10 represent these sub-processes and this is due to the lack of support for this process interaction formalism.

It is not difficult to understand that the goal of claiming route segments is to prevent collision between different vehicles. Speed and position of these side loaders were conceptualized as second order differential equations. Our Java based simulator did not support continuous simulation in which the time advances based on a constant Δt . The lack of continuous simulation forced us to estimate the

The lack of support for the process interaction formalism and support for continuous simulation stipulate a need to support multiple formalisms.

Fig. 2.10: Partial class diagram of Park-A-Car



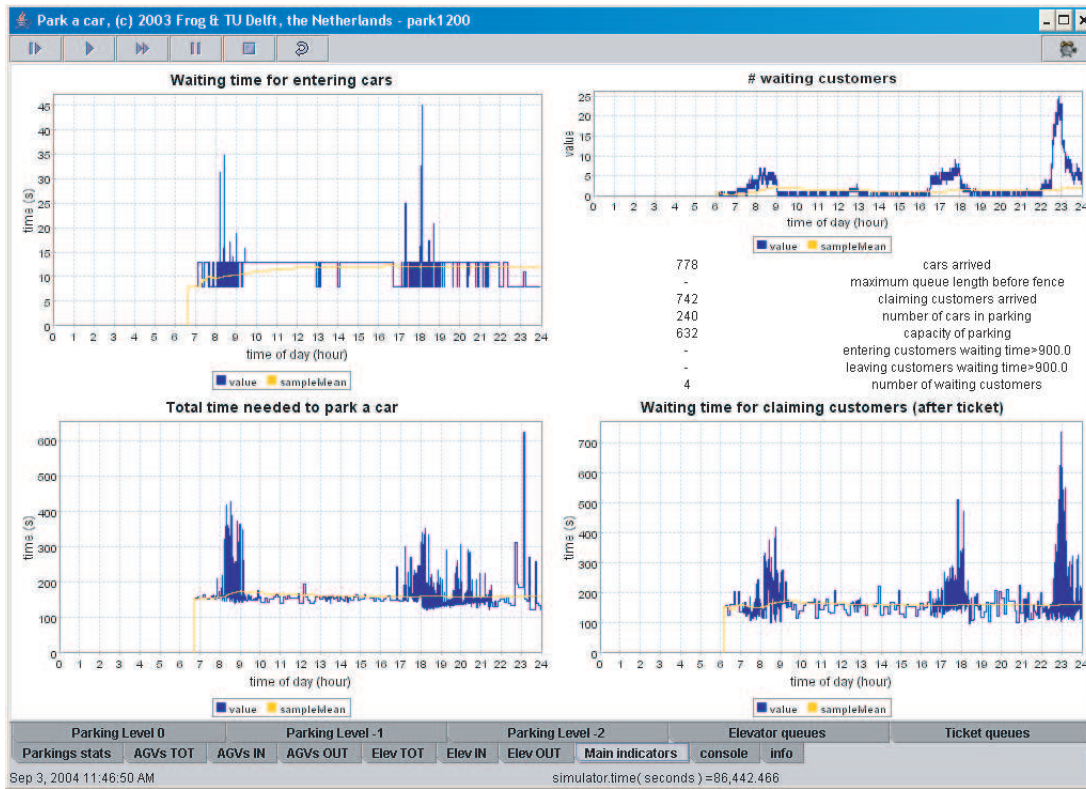


Fig. 2.11: Statistical output of the simulation model

continuous movement of the AGVs; this was achieved by introducing an tailored numerical integration inner-class called `PostponedMovement`.

The lack of support for the process interaction formalism and support for continuous simulation stipulate a need to support multiple formalisms in the specification of a simulation model; this remains a topic for further research.

The web-based animation of the Park-A-Car simulation model is presented in figure 2.8. One can distinguish the driving parking shuttles, the red colored claimed semaphores and a number of parked cars.

The graphical output of the model is presented in figure 2.8. We see 8 elevators as crosses in the center of this figure. On both sides we see AGVs some of which are moving. The ones that are moving are accompanied by red-colored rectangles. These rectangles represent claimed semaphores, or tokens. Once such a semaphore is claimed, other AGVs may not claim it, which prevents collision. The small rectangles represent cars.

The statistical output of the model is presented in figure 2.11, the waiting times for arriving and departing motorists are illustrated in this figure. The Park-A-Car

simulation model furthermore provided charts on elevator usage, individual shuttle movements and motorist waiting times at the ticket counters. We see bottlenecks in the system at 8am, 5pm and 11pm. The long waiting time (≈ 12 minutes) at 11pm is the result of people claiming their car after they have visited a nearby theater.

The municipality of Rotterdam decided based on the simulation model not to implement the option as presented.

The main conclusion drawn from the simulation model was that the requirement of not to exceed 15 minutes waiting time could only be met under optimal circumstances, since neither the AGVs, nor the elevators have spare capacity in the peak hours. The municipality of Rotterdam decided, based on the simulation model, not to implement the option as presented.

2.2.5 Conclusions

The conclusions of this case were...

We introduced our research problem in section 1.5 on page 5 by stating that most current simulation environments are based on an *1-1-1* paradigm; they are designed to serve 1 problem owner, on 1 computer with 1 operating system, 1 formalism. In this section we evaluate to what extent we were able to overcome this paradigm and to provide more useful and usable decision support. Our main conclusions are:

...that vehicle movement in the model is too complex to model using a discrete event formalism,...

- as introduced in section 2.2.2 the behavior of the automatic guided vehicles, i.e. the shuttles, is conceptually described as a set of n^{th} order differential equations. Since Java based simulation model did not support continuous simulation, these functions could only be estimated by the discrete event model specification. We conclude that to improve the usefulness of a simulation environment for the model builder there is a clear need to support both continuous and discrete models.

...that the specification of a model in an object-oriented language is important and...

- with respect to the usefulness of the suite we conclude that the specification of a model in a well understood and supported object-oriented language such as Java is very important. In contrast to prior projects, Frog's engineers were able to verify the model because they understood the language in which the model was specified. Although this conclusion might seem to contrast with the conclusion drawn from the Air Force case in which we concluded that the model structure evaporated, we must understand that this evaporation was the result of a distributed model specification that did not underly the Frog case. We conclude that standardized, i.e. object-oriented, insight into the model directly contributed to the usability and usage of the environment for the model builder.

- where traditional simulation environments restrict the use of concurrent versioning systems, documentation standards and conceptual modeling environments, the Java based suite enables the use of state-of-the-art software engineering tools and techniques. Model specification was no longer tied to one computer but dispersed over multiple geographically distributed locations and developers. We thus conclude that the Java based suite performed better with respect to useability.

...that we were far better equipped for collaborative simulation model specification than traditional environments.

The overall conclusion is that although the suite had a good potential to become a high-quality simulation suite, the current implementation mainly lacked effectiveness with respect to its usefulness. Support for multiple formalisms and linkage with other services, e.g. optimization and GIS services, remained to be established.

2.3 Research question revised

Explorative case studies, we have demonstrated that there is a need for a simulation suite that supports distributed, multi-actor decision making. Using a general purpose programming language, there is a value to be gained from developing a simulation suite; in both case studies the requirements were largely fulfilled and as such both case studies were successfully completed. We nevertheless conclude that the set of created simulation services is far from complete, and that the value of distributed interacting services has not been demonstrated. Model boundaries and model oversight evaporated because of the loosely coupled structure of the environment.

Based on these case studies we conclude there is value to be gained from developing a simulation suite. This resulted in a set of revised research questions.

We formulated the following set of revised research questions based on the conclusions of our explorative case studies.

Research question 1 *Can we create a simulation suite which takes full advantage of the distributed, service oriented computing paradigm?*

Research question 2 *Can we create a simulation suite which supports conceptual modeling freedom using discrete and continuous simulation models?*

Research question 3 *Can we create a simulation suite based on a loosely coupled structure between a model and its environment, e.g. animation, statistics, optimization components?*

Research question 4 *Can we provide scientific evidence that such a simulation suite provides more effective decision than simulation environments that are based on a 1 – 1 – 1 paradigm?*

3. SYSTEMS ENGINEERING PRINCIPLES

Systems engineering and its perspective on information system design form the central themes of this chapter. Systems engineering is distinguished from other fields of engineering in the first section of this chapter. We continue with the principles of designing system architectures in section 3.2. *Object-orientation* is introduced as a preferred modeling paradigm of systems engineers in section 3.3. Principles for object-oriented modeling are discussed in section 3.4.

Systems engineering and its perspective on information system design form the central themes of this chapter.

3.1 *Systems engineering*

Since the 1950s a number of interrelated intellectual areas such as general systems theory, information theory, cybernetics, control theory and mathematical systems theory have emerged (Ashby, 1956; Klir, 1985). These areas can be identified by the general term *systems sciences* of which the engineering subset is called *systems engineering*. Simon (1976); Klir (1985) use three basic components of scientific inquiry to compare system sciences and traditional sciences; these are:

- the *domain* of inquiry
- the body of *knowledge* regarding the domain
- a *methodology* linking activities in the process of problem solving, or knowledge acquisition

We start with a definition of a system as a way to introduce the domain of inquiry of systems engineering, or science.

Definition 3.1.1 *A system is a part of the world we choose to regard as a whole, separated from the rest during a period of consideration, which contains a collection of objects, each characterized by a selected set of attributes, operations and relations (Holbaek-Hansen, 1975).*

A system is a part of the world we choose to regard as a whole, which contains a collection of objects.

The value of this definition for our research is that it is rooted in an activist philosophy; system boundaries, objects and attributes are all subjectively chosen and selected. Systems engineering is thus considered to be a subjective, procedural rational activity.

An *information system* in juxta position to a *real system* is introduced in figure 3.1. Brussaard and Tas (1980) define this *real system* as those parts or aspects of reality we want to investigate as a whole, with the intent to know, or eventually to control.

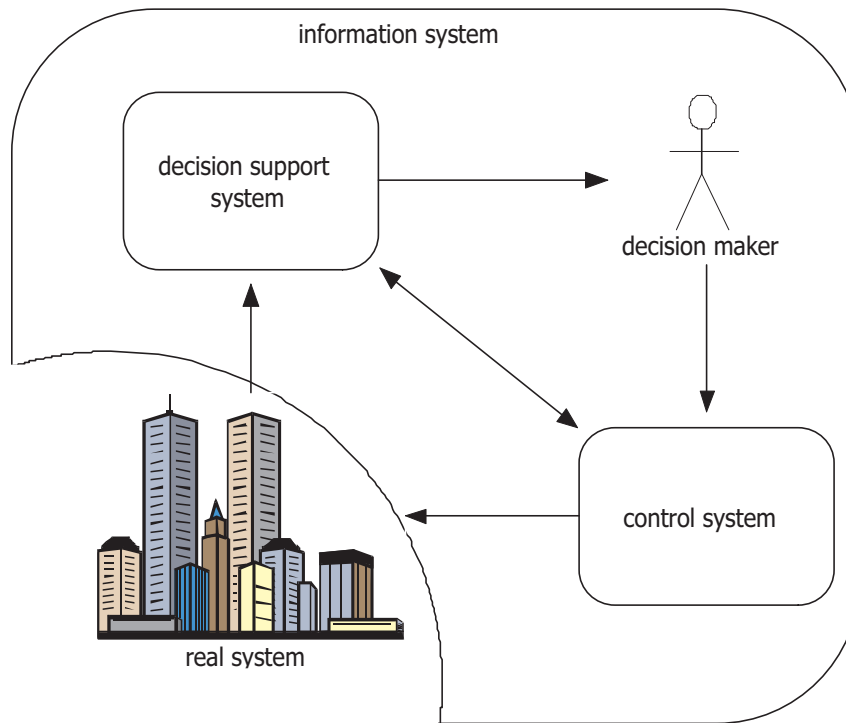


Fig. 3.1: Information System

Brussaard and Tas (1980) continue by defining the functions of *information systems* as: the collection, storage, processing, retrieval, transmission and distribution of data by human beings and machines.

We started this section by introducing the three components by which Simon (1976); Klir (1985) compare different sciences. The first component, i.e. the domain of inquiry, of *systems engineering* can now be presented. According to Klir (1985); Zeigler et al. (2000), systems engineering deals with the design of systems; systems engineers focus on the design and the specification of system structure: the objects, and relations, i.e. the behavior.

The second component by which Simon (1976); Klir (1985) compare different sciences comprises the *body of knowledge* of systems engineering. How, in other words, does one obtain knowledge about the relations within a system? As argued in chapter 1, one can obtain knowledge using a formal mathematical deductive

approach or by using an inductive approach. A consecutive multidisciplinary inductive approach is prescribed by Churchman (1971); Bosman (1977); Sol (1982); Keen and Sol (2005) in the context of decision support for ill-structured problems, in this approach the computer becomes the *laboratory* for the systems engineer.

As in any science, modeling paradigms and languages exist to make the body of knowledge communicatable. Object-orientation has emerged as the de-facto modeling paradigm in systems engineering (Booch et al., 1999). Since information system development has been influenced so heavily by this paradigm, the rest of this chapter is used to introduce its history, principles and consequences.

This is not to forget the third component by which Simon (1976); Klir (1985) compare different sciences: the *methodology*. The methodology of systems engineering forms the basis of chapter 4 on *simulation*. It involves the activities of conceptualization, specification, verification, validation and experimentation.

Object-orientation has emerged as the de-facto modeling paradigm of systems engineers.

3.2 Principles for system design

We present a variety of strategies to divide systems into modular sub-systems in this section. We introduce the value of this concept by taking a look at the size of a typical information system: between 10^1 and 10^5 objects are related (Eckel, 2000). If we cannot find a strategy that we can use to divide such design into sub-systems, (re)use, validity and future development becomes at least questionable. The concept of subsystems is introduced to help us understand the variety of strategies that might be used.

Definition 3.2.1 *A subsystem is a system that is part of some larger system (Gove, 2002).*

This implies a recursive definition of a system S as a set of systems $\{s\}$. The usefulness of this concept is entwined with the concept of *modularity*. The following principle reflects strategies for dividing systems into subsystems.

System design requires decomposition into either vertically partitioned or into horizontally layered subsystems.

Principle 3.2.1 *System decomposition results into either vertically partitioned or into horizontally layered subsystems.*

Both approaches are illustrated in figure 3.2. A *horizontally layered* system is an ordered set of subsystems in which each of the subsystems is built in terms of the ones below it. A *vertically partitioned* system divides a system into multiple autonomous, and therefore more loosely coupled subsystems, each providing a particular service. The orthogonal decomposition of systems into either vertical partitions or horizontal layers is not exclusive. Systems can be successfully decomposed into various combinations according to both approaches; partitions can be layered and layers can be partitioned.

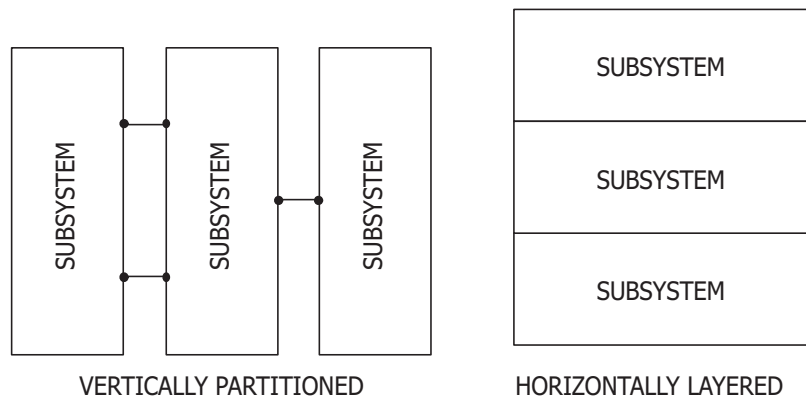


Fig. 3.2: Decomposition of systems into subsystems

This orthogonal decomposition is not exclusive: partitions can be layered and layers can be partitioned.

The value of dividing a system into horizontal layers and vertical partitions is entwined with the concept of *separation of concerns*, and thus with the concept of a subjective choice for a part of a larger system under investigation (Holbaek-Hansen, 1975; Sol, 1982). Separation of concerns is at the core of systems engineering. It refers to the ability to identify, encapsulate, and manipulate those parts of a system that are relevant to a particular concept, goal, task, or purpose (Tarr and Ossher, 2001). Guided by this principle of systems decomposition, we can describe the objects and underlying relations of these decomposed systems: we introduce object-oriented systems description.

3.3 Object-oriented system description

Nygaard and Dahl launched a project in 1962 to develop a discrete event simulation language, to be called *Simula* (Dahl, 2002). The resulting language was called *Simula 1* and was, partly because of European patriotism, based on the *Algol 60* language.

In 1967, Hoare (1968) proposed the concept of *record handling*, consisting of record classes and subclasses. Based on this proposal, Nygaard and Dahl stripped *Simula 1* of all references to simulated time to give us the general purpose *Simula 67* programming language. Nygaard and Dahl used the terms *class* and *object*, and thus object-orientation was officially born.

The use of objects distinguishes object-orientation.

The use of objects distinguishes object-orientation from techniques such as traditional structured methods, i.e. process-based methods in which data and function are separated, or other techniques such as knowledge based systems, i.e. logic programming, or mathematical methods, i.e. functional programming.

Several more object-oriented modeling languages appeared in the mid 1970s as systems engineers began to experiment with this alternative approach to analysis and design. New generations of tools and techniques emerged among which were Fusion, Coad-Yourdon, OMT and OOSE (Meyer, 1997).

A critical mass of ideas emerged in the early 1990s when the designers of these tools and techniques began to adopt ideas from other workers. With this came the advent of the *Unified Modeling Language* (Booch et al., 1999).

Dahl (2002) argues that the importance of the object-oriented paradigm today is such that one must assume that something similar would have come about with or without the Simula effort. The fact remains however, that the object-oriented paradigm was introduced in the mid 60s through *Simula 67*.

The basic theory of object-orientation is to divide a system into objects and relations. As described, an object is characterized by a selected set of attributes, operations and relations. Objects are instances of a *class*, which is a description of a set of objects that share the same attributes, operations, relationships and semantics (Booch et al., 1999). Object-orientation distinguishes the following two types of relationships:

- *generalization* \leftrightarrow *specialization*; class *A* is a generalization of class *B* if, and only if, every instance of class *B* is also an instance of class *A*, and there are instances of class *A* which are not instances of class *B*. Equivalently, class *A* is a generalization of *B* if *B* is a specialization of *A*. Formally we speak of a generalization between classes whenever $\forall x(Bx \rightarrow Ax), \exists x(Ax \wedge \neg Bx)$.
- *association*; where *generalization* specifies a relation between classes, *association* refers to the structural relation between objects, or instances. A special form of association that specifies a *whole-part* relationship between the aggregate, i.e. the whole, and the object, i.e. the part, is called *aggregation*, or *decomposition*. An aggregation relation is an association relation with the exception that instances cannot have cyclic aggregation relationships, i.e. a part cannot contain its whole.

The basic theory of object-orientation is to divide a system into objects and relations, i.e. ...

...generalization and association.

Both types of relations are presented in figure 3.3. The arrow connecting the **Manager** and the **Employee** illustrates a specialization relation. A manager is thus a special employee and its class inherits both name and contract from the **Employee** class.

The association between the **Manager** class and the **Employee** class distinguishes the **Manager** from the **Employee** and therefore justifies the existence of the **Manager** class. This relation is equivalent to the relation between an employee and his or her name and contract. They all express association relations.

A number of principles, or guidelines, have evolved to improve the quality of object-oriented modeling and design since its birth in 1967.

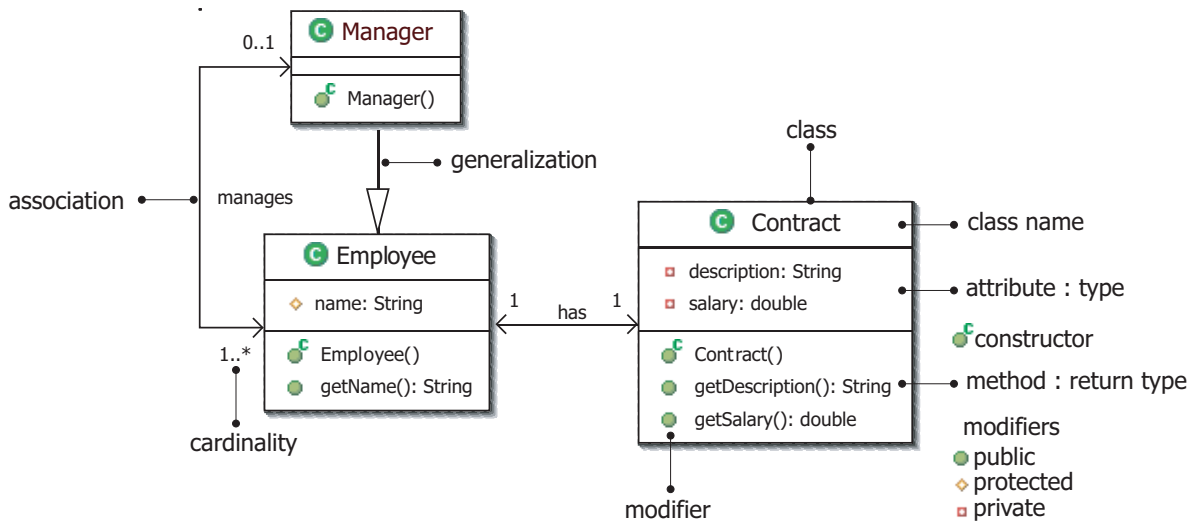


Fig. 3.3: Relations in object-orientation

3.4 Principles for object-oriented modeling

A number of principles, or guidelines, have evolved to improve the quality of object-oriented modeling.

The principles for object-oriented modeling as described in (Lee and Tepfenhart, 2002; Booch et al., 1999; Eckel, 2000) are presented in this section. Where these principles require examples to illuminate their inner-works or importance, they are specified in the Java programming language.

Principle 3.4.1 *Class design: the ability to specify classes.*

A class can be viewed from different perspectives: modeling, design, implementation and compilation. From a modeling perspective, a class is a template for a *category* of objects. It defines the attributes, operations and relations of the category and thus of all objects belonging to the category. From an implementation perspective a class is a *global* object with *globally accessible* attributes, relations and operations.

Information hiding denotes that an object explicitly describes which attributes and methods are publicly visible.

Principle 3.4.2 *Information hiding: an object explicitly describes which attributes and methods are publicly visible and therefore accessible to other objects.*

Object-orientation provides modifiers to define to what extent attributes and operations are hidden from other objects. Besides a public modifier, an object may declare its attributes to be *protected* or *private*. Where the private modifier is used to encapsulate the attributes of an object, the protected modifier grants access to the object and all its subclasses (see principle 3.4.5).

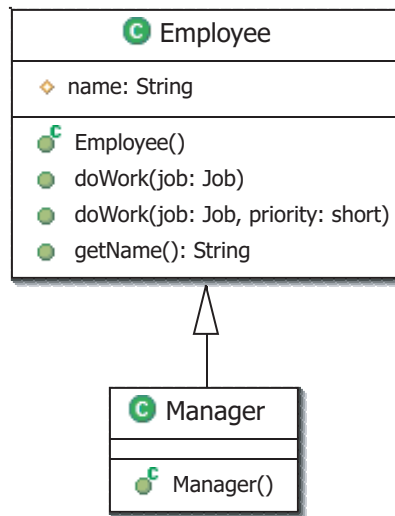


Fig. 3.4: Overloading

Principle 3.4.3 Encapsulation: *attributes and operations uniquely belong to an object.*

Encapsulation denotes that attributes and operations uniquely belong to an object.

Encapsulation is a concept of which the value might not be immediately apparent. Its importance results from the elimination of the need to check that attributes are manipulated by an appropriate operation. The manipulation of an attribute is explicitly granted to the object that owns the attribute.

Principle 3.4.4 Polymorphism: *the word polymorphism comes from the Greek for "many forms" and denotes an object's capacity to have multiple forms.*

Polymorphism denotes an object's capacity to have multiple forms.

Cardelli and Wegner (1985) divide polymorphism into two categories, i.e. runtime and universal polymorphism. Runtime polymorphism can be further categorized into *coercion* and *overloading*. Universal polymorphism includes *parametric polymorphism* and *inclusion*. These four variants can be defined as follows:

- *coercion* represents an implicit parameter type conversion to the type expected by a method or an operator, thereby avoiding type errors. A good example is $2.0+2.0$ versus $2.0+"2.0"$. Where two double values are added in the first examples, coercion converts the 2.0 double value of the second example into a string and the result will be the concatenated string "2.02.0".
- *overloading* permits the use of the same operator or method name to denote multiple, distinct program meanings. An example of overloading is presented in figure 3.4. In this example we define two methods, both are

named `doWork()`. The method, and thus the implementation to be used, depends on whether a priority is given as argument to the invocation.

- *parametric polymorphism*, also referred to as *type parameterization* or the use of *generics*. Lisp and Simula 67 were the first languages to support parametric polymorphism (Eckel, 2004b). One may define *latent types*, or *latent classes*, in these languages, i.e. a type that is implied by how it is used, but that is never explicitly specified. That is, the latent class is implied by the methods that one may call on it. If one calls methods `f()` and `g()` on a latent class, then one implies that a class has methods `f()` and `g()`, even though that class is never actually defined anywhere (Eckel, 2004b,a).
- *inclusion* achieves polymorphic behavior via an inclusion relation between classes. The inclusion relation is a subtype relation in most object-oriented languages; inclusion is therefore often called *subtype polymorphism*.

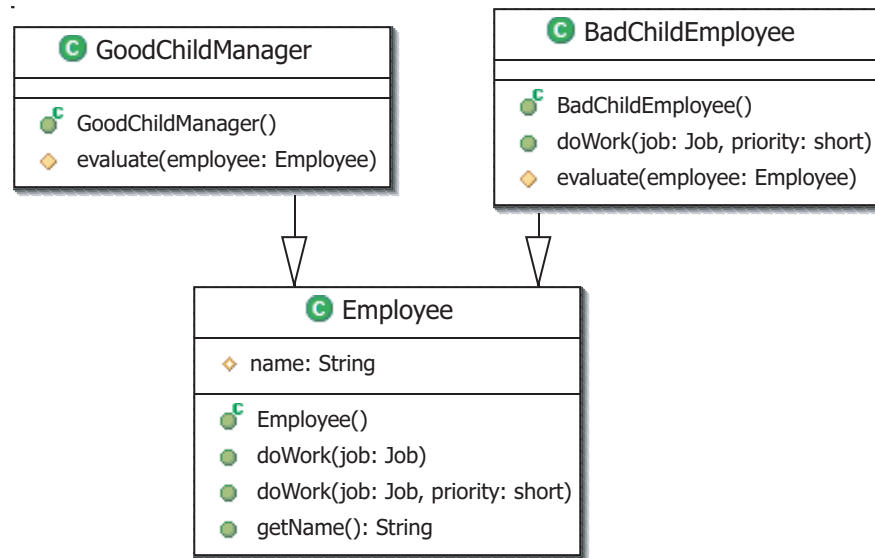


Fig. 3.5: Good child versus bad child

Inheritance denotes that classes can be organized in a hierarchical structure.

Principle 3.4.5 *Inheritance: classes can be organized in a hierarchical inheritance structure. In such a structure, the subclass inherits the protected and public attributes and operations from the superclass.*

The value of inheritance is far from trivial. Where some argue that inheritance should be the leading principle in systems design, others argue that its value is

overrated. As presented in section 3.3, inheritance embodies the specialization relation between classes. An *abstract class* is used to create only subclasses; therefore there may be no direct instances of such class. The principle of inheritance has the following properties:

- object instances of the descending class have values for all the attributes and relations of the ancestor class.
- all operations provided by the ancestor class must also be provided by the descendent class. *Visibility* and accessibility of operations may not be limited by subclasses.
- inheritance distinguishes *good children* versus *bad children* (Eckel, 2000). A good child is a descending class without polymorphism; all operations provided by the ancestor are used by the descendent class. A bad child supplies its own customized implementation for some of the operations provided by the ancestor class. The difference between a good child and a bad child is illustrated in figure 3.5: where the good child inherits the `doWork` methods of the `Employee`, the bad child overwrites its implementation
- inheritance is *antisymmetric*. If class A is a subclass of class B, then class B cannot be a subclass of class A. Inheritance is furthermore *transitive*. If class A is a subclass of class B and class B is a subclass of class C, class A is also a subclass of class C.

Principle 3.4.6 *Delegation: an object passes the invocation of an operation on to another object which actually fulfills the invoked operation.*

Delegation denotes that an object has passed the invocation of an operation on to another object.

Delegation is also referred to as the *perfect bureaucratic principle*; an invoked operation is delegated from an object to another object that has the attributes and operations to fulfill the required operation. Delegation is closely related to the *design by contract* principle (see principle 3.4.9). Delegation implies that it is the authority that is delegated; not the responsibility.

Since the early 1980s there has been much debate over which principle embodies a more powerful concept for implementing the specialization relation presented in section 3.3: the principle of inheritance or the principle of delegation. Stein, Lieberman and Unger came together in 1987 to discuss their differences and this resulted in a statement reflecting the need for both principles (Lieberman et al., 1988). This treatment became known as *The Orlando Treaty*.

Asynchronous communication is the principle of invoking an operation on another object where the requesting object does not expect an immediate result.

Principle 3.4.7 *Asynchronous communication: an object invokes an operation on another object where it does not expect an immediate result (Booch et al., 1999).*

This principle introduces the pattern of *subscription*. Instead of polling an object for a state change in which one is interested, an object provides operations for asynchronous subscriptions. Whenever the state change occurs, the object that subscribed is notified. The object interested in potential state changes is referred to as *Listener*, the object which accepts asynchronous subscriptions is called *Producer*.

Late binding is the support for the ability to determine the specific class, and thus the specific specification of an operation, at runtime.

Principle 3.4.8 *Late binding: support for the ability to determine the specific class, and thus the specific specification of an operation, at runtime.*

Although the concept of late binding increases the complexity of understanding object-oriented code, it is one of the most powerful methods used in the pursuit of loosely coupled systems. The concept is illustrated by the following example:

```
26 public class Worker
27 {
  ..
33   public void execute(final Worker worker)
34   {
35     worker.execute(worker);
36   }
  ..
37 }
```

At first sight, the above specification of the `execute` operation might seem dubious and destined to produce an infinite loop; this is not the case. Late binding allows us to invoke the `execute` operation with an instance of a subclass of `Worker`. Then this overridden implementation of the `execute` operation is invoked.

Current object-oriented programming languages such as Java even allow objects to automatically download unavailable class information of such subclasses at runtime. This is called *dynamic class downloading* (Sing, 2000).

Design by contract is the ability to design a set of operations as a contract.

Principle 3.4.9 *Design by contract: the ability to design a set of operations as a contract.*

Meyer (1992) originated a design principle called design by contract: in addition to specifying the signature of a method, the designer also specifies the pre-conditions, the post-conditions and the invariants, i.e. conditions that should be true of a class

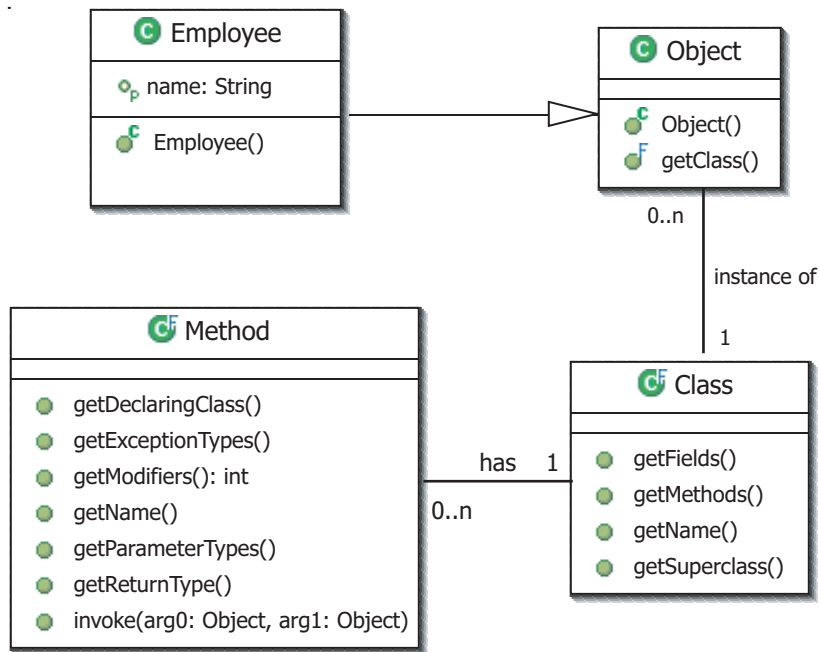


Fig. 3.6: Reflection

in general. The strength of this principle is that it gets the designer to think clearly about what service an method provides (Meyer, 1992).

A few programming languages, e.g. Eiffel, implement pre- and post-conditions in executable code so that they are checked at run time. Most programming languages do not have such support, so programmers who want to use pre- and post-conditions often write comments documenting the conditions. In these languages the principle of design by contract is embodied in the signature of a method, which is published in an *interface*. An interface describes the syntax of a service description; it specifies the signatures of the operations to be implemented by classes implementing the interface. If we recall the service oriented computing paradigm of section 1.6.2 we may conclude that an interface embodies the concept of a service description. Thus interfaces promote the conceptual strength for separating requirements from specification, and increase our ability to design loosely coupled systems.

Principle 3.4.10 *Reflection: an object knows the detailed information about the class(es) and interface(s) of which it is an instance.*

A consequence of reflection is that an object can, at runtime, acquire detailed information on its state and methods. The principle of reflection is presented in figure

Reflection is the ability of an object to know detailed information about the class(es) and interface(s) of which it is an instance.

3.6. The `Employee` class inherits a method `getClass()` which returns the class of which the object is an instance, i.e. `Employee.class`. This `Employee.class` contains methods to resolve methods and fields.

3.5 *Summary*

Systems engineering was presented in this chapter, where we presented a system as a part of the world we choose to regard as a whole, separated from the rest during a period of consideration, which contains a collection of objects, each characterized by a selected set of attributes, operations and relations.

The de-facto language to describe systems is the object-oriented description. Although object-orientation only specifies two orthogonal types of relations, object-oriented system design is considered to be complex. In this chapter we presented a set of design principles which will form the basis for the design presented in the remainder of this thesis.

4. SIMULATION AS A METHOD OF INQUIRY

4.1 *Actors and activities in a simulation study*

Decision support is not just about software, models and tools: decision support is about making decisions following a method of inquiry. Simulation is the preferred method of inquiry in the context of ill structured problems (Shannon, 1975; Sol, 1982).

Shannon defines *simulation* as the process of designing a model of a real system and conducting experiments with this model for the purpose of either understanding the behavior of the system or of evaluating various strategies for its operation (Shannon, 1975). The following *actors* can be distinguished in the more general domain in decision support (Keen and Sol, 2005):

Simulation is the process of designing a model of a real system and conducting experiments with this model.

- *stakeholders* make the decision and are committed to the results. Since stakeholders are most often senior members of an organization, they commonly do not create the models themselves and are merely interested in the output of a rational inquiry, both substantive and procedural.
- *non-stakeholders* are involved in the process of decision making, but have no stake in its outcome. Consultants, supporting staff, system administrators and the actual model builders, are often non-stakeholders.

A further distinction is made with respect to the role of an actor. The following roles can be distinguished with respect to decision support systems:

- *users* are considered to be customers of the system. Users access the simulation suite to use it to experiment and analyze scenarios.
- *builders* have a more intensive role. Builders conceptualize and specify a system under investigation. This role is most often played by consultants. A builder is thus involved in all the steps of the modeling life cycle.
- *systems engineers* design decision support services, e.g. a simulation suite. Although the engineers need not to be involved in any of the steps of an actual modeling cycle, they need to be able to understand the requirements of and have experience with the roles of users and builders.

- *maintainers* deploy and maintain simulation tools within the enterprise information architecture. Maintainers have thus stake in the quality aspects of a simulation suite.

One of the cornerstones of the studio based approach to decision support is that the above roles are neither exclusive, nor static. Actors can have more than one role and an actor's role may change over time. One might then ask why do these roles need to be delineated, to which the answer is: the roles are used to provide a rational structure for the identification of requirements for the simulation suite. In short roles are used to express *who* needs *what*. Before we elaborate on the activities involved in a simulation study, we introduce a formal framework containing the concepts involved.

4.2 A framework for simulation

A framework for simulation based on Zeigler's "*Theory of modeling and simulation*" (Zeigler et al., 2000) is presented in this section. The framework is used to formalize the concepts and terms of a simulation study. See figure 4.1 for a graphical representation of the framework and its basic entities.

- The *real system* is that part of reality which is under consideration. It is viewed as the source of observable data. Zeigler refers in the definition of this system to Klir's epistemological level 0 (Klir, 1985). The real system is what Zeigler addresses as the source system. The real system reflects the definition of a system as presented in 3.1.1 on page 35: it is a part of the world we choose to regard as a whole, which contains a collection of objects and underlying relations (Holbaek-Hansen, 1975).
- The *experimental frame* consists of a specification of the conditions under which the system is experimented with or observed. The experimental frame is discussed in section 4.5.
- The *simulation model* is a system description, e.g. an object-oriented description or a mathematical description, of a real system. A simulation model specifies both the relations and the behavior of the system as a function of the system time.
- The *simulator* is an object, not part of the real system, that is used to generate the behavior of the simulation model.

A simulation model is a system description following.

The simulator is used to generate the behavior of the simulation model.

The value of figure 4.1 for our research is the distinction made in this framework between the simulator and the simulation model. While most simulation software

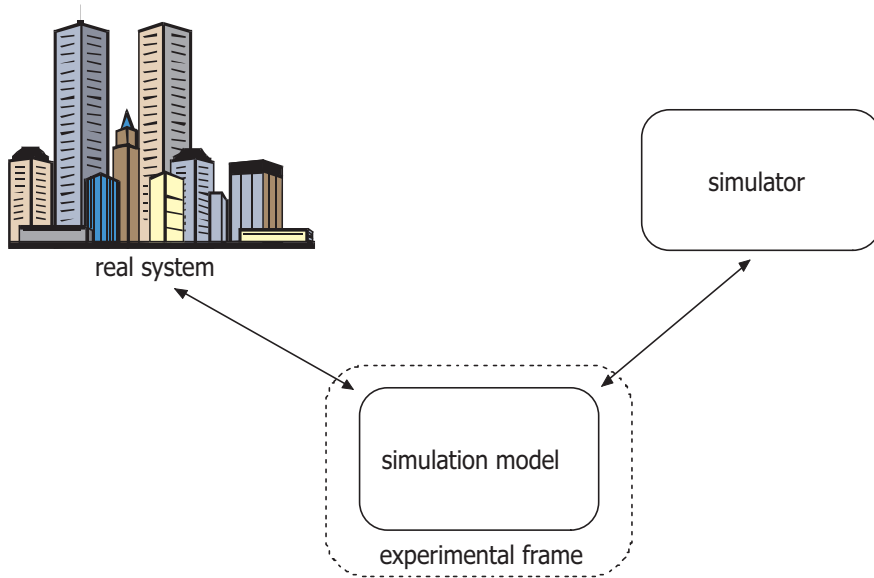


Fig. 4.1: A framework for simulation

environments do not uphold this distinction (Hlupic, 1993), we argue in this thesis that there is a lot to be gained from separating these parts. Advantages include the ability to use different simulators on the same model and the ability to (re)use a model in non-simulated operational business processes, see chapters 7 and 8. Before we elaborate on the specification of a simulation suite in chapter 5, we need to present several conceptual modeling formalisms.

4.3 Multi-formalism modeling

As introduced by Kiviat (1967); Fischman (1973) simulation models have two orthogonal types of structure: a *static* and a *dynamic* structure; the dynamic structure is also referred to as the *behavior* of a simulation model. Where the static structure represents the *state* of the simulation model, the behavior represents its *time-dependent* state transitions.

Nance (1981) introduces a small set of basic definitions in which he carefully distinguishes time and state relationships. He argues that any simulation model representation can be constructed with this given set of definitions. Nance's starting point is the object-oriented system description presented in section 3.3; a simulation model is thus considered to be comprised of objects described in terms of their attributes and values. Assigning a value to an attribute of an object in a system description is done based on observations. These observations may change over

Simulation models have two orthogonal types of structure: a *static* and a *dynamic* structure

the actual state of an object. They thus may change over time, e.g. a patient's temperature, over place, e.g. the location of an airplane. Such an underlying property, used to distinguish different observations of the same attribute, is called a *backdrop* (Klir, 1985).

In time-based simulation, *system time*, or *simulation time*, is used as the backdrop of the system. Nance presents the definitions illustrated in figure 4.2 concerning system, or simulation, time. An *instant* is defined as a value of system time at which the value of an attribute can be altered. An *interval* is the duration between two successive instants and a *span* is the concatenated succession of intervals.

In time based simulation, system time is used to distinguish different observations of the same attribute.

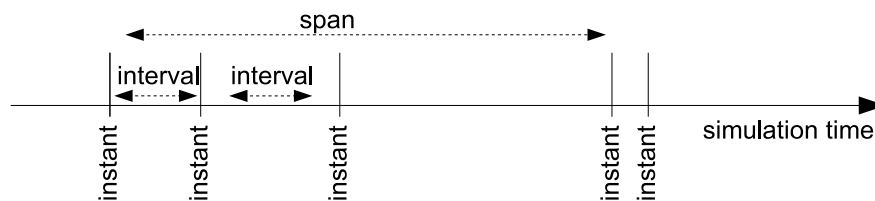


Fig. 4.2: Time related concepts

The *state* of an object is the enumeration of all attribute values of an object at an instant. The relation between system time and system state are illustrated in figure 4.3. An *event* is a change in the state of an object at an instant. A *process* is the succession of states of an object over a span.

There are different approaches that can be used to describe the behavior of a simulation model; they are referred to as a *world view*, *formalism*, or a *modeling construct*.

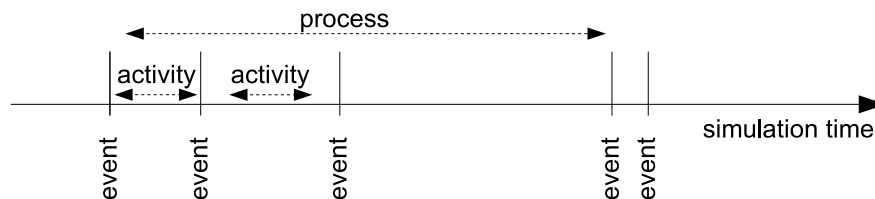


Fig. 4.3: State related concepts

While this set of definitions describes the static structure of a simulation model, there are different approaches that can be used to describe the behavior of a simulation model. These approaches are referred to as a *world view*, *formalism*, or *modeling construct*.

The evolution of a formal description of simulation formalisms started with Zeigler's categorization in Zeigler (1976). Zeigler presents formalisms based on the continuous versus discrete nature of their time advancing and state transition functions. This results in the following fundamental formalisms.

- *Differential equation system specification* (DESS): this formalism represents the traditional differential equations with a continuous time advancing function and a continuous state transition function.
- *Discrete time system specification* (DTSS): this formalism represents systems with a continuous state transition function and a discrete time advancing function, i.e. $t(x_n) = t(x_{n-1}) + \Delta t$.
- *Discrete event system specification* (DEVS): this formalism represents systems which operate on a discrete time function with a discrete state transition function.

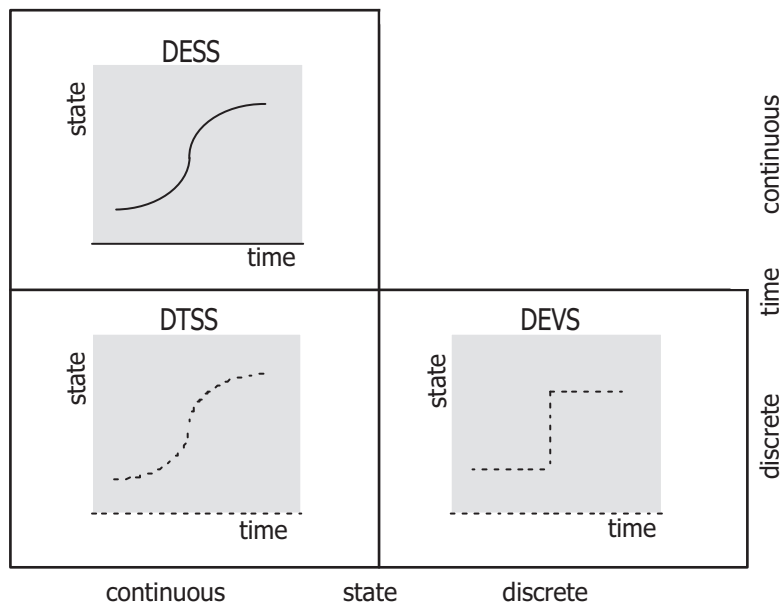


Fig. 4.4: Continuous versus discrete formalisms (Zeigler et al., 2000)

Zeigler et al. (2000); Vangheluwe and de Lara (2002) argue that systems often have components and aspects for which the state transition function cannot be described in a single comprehensive formalism. It is desirable to express state transition functions as a function of multiple formalisms for the design and analysis of such systems; hence the concept of *multi-formalism modeling*.

Vangheluwe and de Lara (2002) present the formalism space in what is known as a formalism transformation graph, see figure 4.5. The different formalisms

It is desirable to express the behavior of a model as a function of multiple formalisms, i.e. *multi-formalism modeling*.

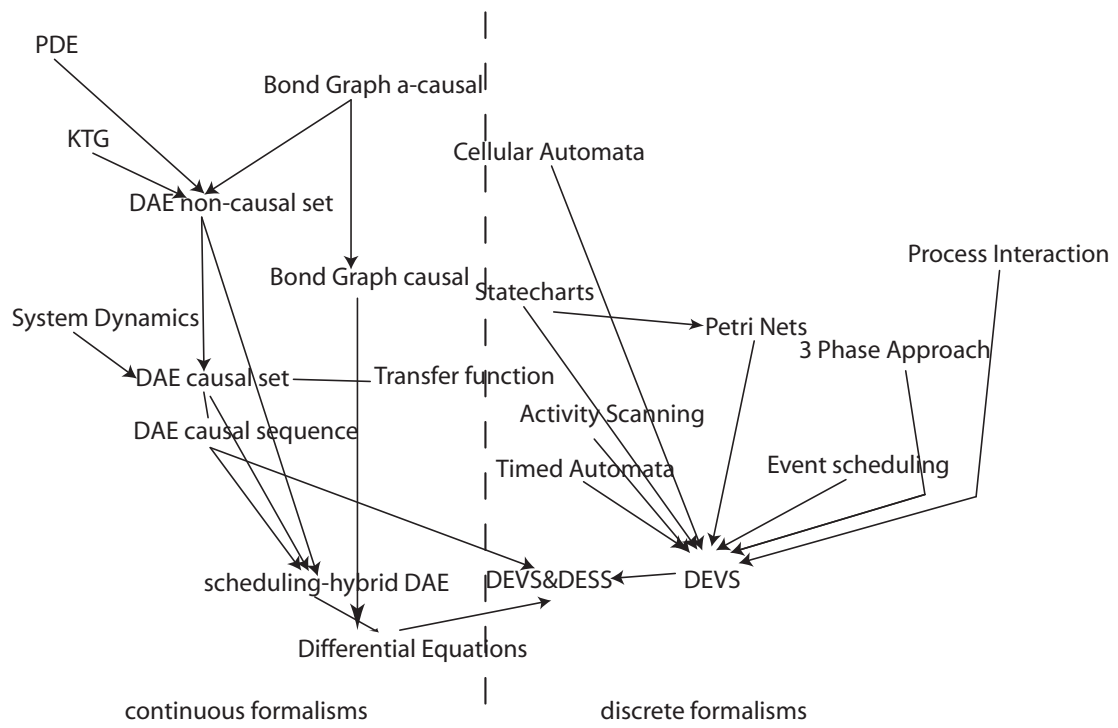


Fig. 4.5: Formalism transformation graph (Vangheluwe and de Lara, 2002)

are presented as the nodes of the graph in figure 4.5. The vertical dashed line delineates the categorization between continuous and discrete formalism. The arrows in figure 4.5 denote behavior-preserving homomorphic relations between formalisms. These relations are also referred to as *embedded* relations in which one formalism is mapped onto another (Zeigler et al., 2000).

Zeigler (1976); Zeigler et al. (2000) conclude that any formalism can be embedded in either the discrete event system specification (DEVS) or in the continuous differential equation system specification (DESS). Zeigler et al. (2000) show that a combined *DEVS* and *DESS* formalism is *closed under coupling* which implies that coupling models expressed within a formalism produces composite models that can also be expressed in the formalism. The main conclusion is that this combined *DEVS* and *DESS* formalism provides us with a formalism in which any time based modeling formalism can be embedded. The graph illustrated in figure 4.5 thus becomes traversable. As shown in the formalism transformation graph of figure 4.5, a multitude of more specialized formalisms has been developed.

In the subset of continuous formalisms these formalisms are related to a specific domain. For example, where system dynamics is targeted at social or ecological systems, bond graphs are commonly used in engineering systems with a variety of thermal, mechanical or electrical components.

The discrete formalisms are not related to a specific domain, in each formalism a unique approach is followed to specify, or group, the behavior of a simulation model. Overstreet and Nance (1986); Page et al. (1997) refer to the concept of *locality* when they speak of grouping behavior in a simulation model.

4.4 Three classical formalisms for discrete event simulation

Three classical formalisms exist to describe the behavior of a system under investigation in discrete event simulation. The meta-model of these formalisms is the selected set of state and time definitions introduced in section 4.3.

The differences between these formalisms is based on how the behavior, i.e. time advancing function, of the simulation model is specified. The three classical formalisms for discrete event simulation are:

- *event scheduling*: a modeler defines events at which discontinuous state transitions occur. An event can cause, via scheduling, other events to occur. As described by Balci (1988) the strategy for the event scheduling world view is to repeatedly select the earliest scheduled event, to advance the simulation time to the execution time of that event and to invoke the operation specified by the event. In the event scheduling formalism the behavior and thus the processing of the simulation model is grouped in a time sorted eventlist: the simulation model is described as a time sorted set of scheduled events.

The combined *DEVS* and *DESS* formalism provides us with a formalism in which any time based modeling formalism can be embedded.

Three classical formalisms exist to describe the dynamic structure of a system under investigation in discrete event simulation.

- *activity scanning*: activity scanning, also known as the *two-phase approach* was first used in the language CSL (Buxton and Laski, 1962). Activity scanning is a form of rule based programming, in which a rule is specified upon the satisfaction of which a predefined set of operations is executed (Balci, 1988). Activity scanning is referred to as the two-phase approach because the simulation model behavior consists of two phases. In phase one the simulation time is increased with a fixed time step, i.e. $t(x_n) = t(x_{n-1}) + \Delta t$. In phase two, the conditions of the activities are tested in the order of activity priorities. If an activity's condition is satisfied, the actions of that activity are performed. State-based and mixed, i.e. time-based and state-based, events are also referred to as *contingent event* (Nance, 1981). Activity scanning is generally considered less efficient with respect to computational execution because of the fixed time step of its time advance function (Balci, 1988).
- *process interaction*: each process in a simulation model specification describes its own *action sequence* (Overstreet and Nance, 1986). This formalism thus reflects the *autonomy* of an individual process, i.e. the life cycle, and the *concurrency* in the execution of distinct processes. A process must have the ability to *suspend* and *resume* operation in its action sequence. A classic example of process interaction is presented by Birtwistle (1979). This example presents a simulation model of boats entering a port and competing for resources. Boats are the objects that trigger processes of this simulation model and define the following sequence: enter port, claim jetty, claim tugs, undock, release tugs, release jetty and leave port. The behavior of the model is thus grouped within the boats, i.e. the temporary objects as they interact with the permanent object during the execution of the simulation model.

A discrete simulation language is said to be either *event-oriented*, *activity-oriented*, or *process-oriented*.

A simulation language is said to be *event-oriented*, *activity-oriented*, or *process-oriented* whenever it supports simulation models which express their behavior according to the *event scheduling*, *activity scanning* or *process interaction* formalism.

The above sections provide us formalisms by which a real system can be abstracted into a simulation model. The next step is to design the experiment. This step is complex and much debated.

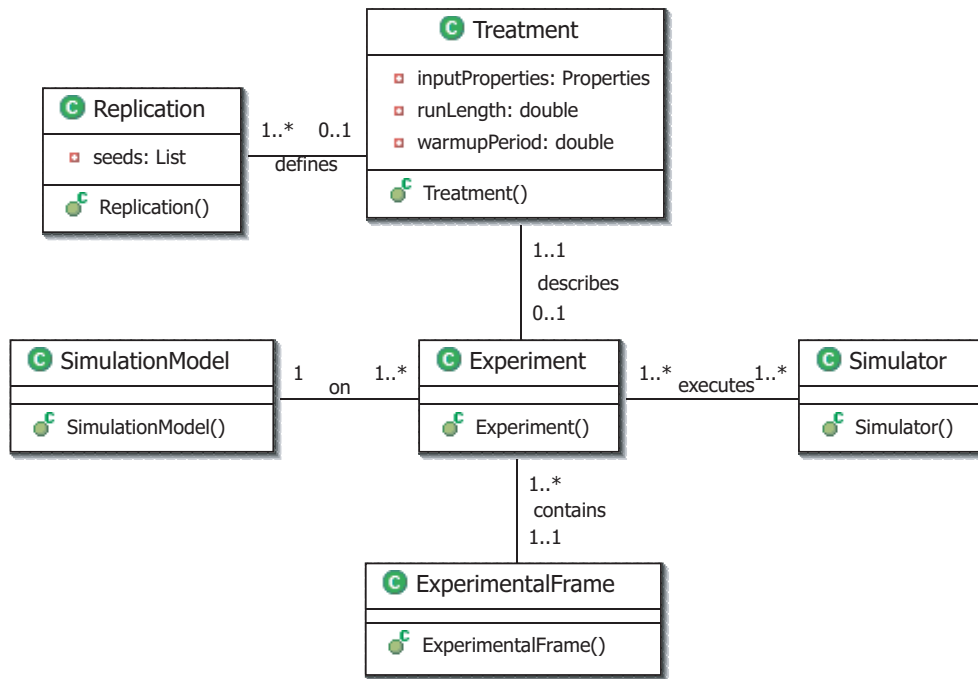


Fig. 4.6: A framework for experimentation

4.5 Experimental design

The mutual relation between experiments and a model is concentrated on in this section. Our aim in this section is primarily to answer a number of questions such as: What is an experiment? What are the constraints imposed by an experiment on model design?

In line with Law and Kelton (2000) we argue that these questions are important because often a great deal of time and money is spent on model development, but little effort is made to analyze the output of an experiment appropriately (Law and Kelton, 2000).

To prevent misuse of tools and techniques, we pay a lot of attention to the concept of an experimental frame. This concept, presented in figure 4.6 follows Ören and Zeigler (1979); Sol (1982); Zeigler (1984) in their specification of an experiment, a treatment, a run control and a replication.

According to Sol (1982) a simulation model is transformed into an executable simulation model system by including provisions to expose it to a *treatment* by which it is placed in an *experimental frame*. The following concepts define a framework for experimentation:

- a *treatment* consists, according to Ören and Zeigler (1979); Sol (1982) of input data, initialization conditions, and run control conditions.
- a *run control* specifies the experimental conditions under which the treatment is conducted on the simulation model, i.e. the runlength and the warmup period (Law and Kelton, 2000). The run control variables are specified as attributes of the treatment (see figure 4.6).
- an *experimental frame*, is defined as a set of possible treatments.
- an *experiment* is a set of replications, with individual pseudo-random number streams, under the same treatment, simulator and model.
- a *replication* is one run out of a collection of runs under the same treatment, except for initialization conditions that provide statistical independence, i.e. the seeds.

4.6 Activities involved in a simulation study

The next step is to identify the activities of a simulation study. Following Shannon (1975); Sol (1982); Banks (1998) we approach simulation as a method of inquiry in which decision makers create a model of a real system, which serves a goal of experimentation, to support actual decision making. The following activities make this method.

- *Model conceptualization*: the abstraction of a real system by a conceptual model (Banks, 1998). In our system of concepts the object-oriented system description forms a central position. Conceptual models are thus object-oriented models representing the objects and relations of the real system under investigation.
- *Model specification*: the collection of empirical data and the specification of attribute values of the objects specified in the conceptual model in a computer-recognizable simulation model. (Banks, 1998). The creation of an experimental design forms part of this activity.
- *Verification*: the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model (Roache, 1998). The initialization conditions, the run control conditions and the number of replications of the different treatments are determined in this activity.

We approach simulation as a method of inquiry in which decision makers create a model of a real system, which serves a goal of experimentation, to support actual decision making.

- *Validation*: the process of determining the degree to which a model is an accurate representation of the real system from the perspective of the intended uses of the model (Roache, 1998). We distinguish several approaches to validation. We refer to *structural validation* if hypotheses on the behavior of the simulation model are checked. We refer to *replicative validation* if values of endogenous attribute values are compared with the ones found in the real system. We refer to *predictive validation*, or *expert validation* whenever the plausibility of a simulation model is tested by experts (Sol, 1982).
- *Experimentation*: the process of using the specified model for the purpose of either understanding the behavior of the system in question or of evaluating various strategies for its operation (Shannon, 1975).

After introducing the theories on simulation in conjunction with the object-oriented approach to the design of information systems, we will now introduce our theory and resulting requirements on the design of a simulation suite.

4.7 Requirements for a simulation suite

The research question was presented in chapter 1 as: Can we create a simulation suite for decision makers that supports a studio-based decision process and improves their performance when solving ill-structured, multidisciplinary problems? We argued that such a suite would better support bounded rationality in the context of ill structured problem solving.

The general concept of a suite and introduce the requirements for its development are presented in this section. An elaboration of these requirements is presented in section 4.8. To emphasize how the concept of a suite differs from traditional, more substantive, simulation environments, we start with its definition.

Definition 4.7.1 *A suite is a well chosen set of services and standards for inter-connectivity; a decision support suite is thus a chosen set of services and standards to support a decision making process.*

The concept of well chosen is related to the requirements of the suite. Before introducing these requirements, we recall the definition of a service as presented in section 1.6.2 on page 7.

Definition 4.7.2 *A service is a self-describing, open component that performs a specific function and is designed to work with other services (Papazoglou and Dubray, 2004).*

Distribution forms the pillar of this research. We argue that the creation of a distributed simulation suite will allow us better to support simulation as a method

Distribution is required in all activities of the simulation model cycle. We thus require distributed simulation model conceptualization, specification, experimentation and evaluation.

of inquiry; distribution enables the involvement of multiple actors in the decision process. Distribution is required in all activities of the simulation model cycle. We thus require distributed simulation model conceptualization, specification, experimentation and evaluation.

Besides this requirement for distribution, we introduce four other requirements; the combination forms the set of requirements for the suite to be developed in chapter 5.

- An inquiry system supporting a model cycle should be expressible in a system of instruments to be able to discuss the implicit premises in a common frame of reference. This requirement is called the *metatheoretical freedom* (Sol, 1982).
- An inquiry system should support the construction of both a conceptual model and a specified simulation model, each potentially in a different language. This is referred to as a *requirement for extensibility* (Ören and Zeigler, 1979), or *conceptualization freedom* (Sol, 1982).
- The inquiry system should support the conceptualization and specification, as well as the solution finding, by iterative analysis and synthesis. This asks for *modeling freedom* leading to consistent model systems that show a good correspondence. The structure laid down in the simulation model should not be constrained by the way the behavior is to be simulated (Sol, 1982). Our notion of modeling freedom goes further: we argue, based on Zeigler et al. (2000); Vangheluwe and de Lara (2002), that modeling freedom requires the freedom of multi-formalism simulation: the freedom to apply different formalisms on different subsystems.
- *Solution finding freedom* refers to the ability to generate solutions by changing the alternative space and the ways this space can be explored in view of human cognitive constraints. The solution finding freedom requires support for insight through animation, report generation and statistical analysis.

In the following section, we will work out these freedoms into a set of more detailed, fine grained requirements that we will use for the development of a simulation suite.

4.8 Requirements worked out

The verb *require* descends from the Latin word *requirere* which means *to seek for*. A requirement embodies the idea of a search by someone for a specific purpose and should answer the basic question *Who needs what, why do they need it?* This *who*, *what* and *why* question is presented in figure 4.7 as a 3-dimensional requirement

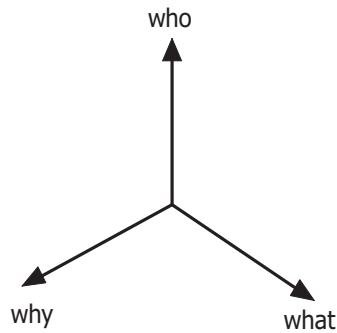


Fig. 4.7: Requirement space

space. We will gradually fill this space with the requirements for a simulation suite in this section.

This description of these requirements is presented in an order that is similar to the structure of this thesis. Requirements for a specific activity and for a specific role are described based on their usefulness. We formulate the specific activity and role with an '*activity* → *role*' label at the beginning of each requirement.

4.8.1 Usefulness

The concept of *usefulness* of decision support tools expresses the value they add to the decision making process. It thus relates to the analytic models, the embedded knowledge and the information resources available in a model or tool. The main requirements supporting usefulness for a simulation suite are:

Requirement 4.1 *specification* → *builder*: a simulation suite is required to provide: pseudo-random number generation, statistical distribution functions, time-flow mechanisms and statistical analysis routines (Nance, 1995).

A simulation suite is required to provide: pseudo-random number generation, statistical distribution functions, time-flow mechanisms and statistical analysis routines.

Requirement 4.2 *specification* → *builder*: a simulation suite should enable the incorporation of domain specific algorithms and libraries for the specification of models. Using domain specific libraries fundamentally increases the embedded knowledge of the system under investigation and thus the usefulness of the suite.

A simulation suite should enable the incorporation of domain specific algorithms

Sol (1982); Zeigler (1976); Zeigler et al. (2000) argue that a simulation language should provide model builders with the freedom to choose a formalism to conceptualize and to specify a system under investigation. Based on the universal formalism transformation graph of Vangheluwe and de Lara (2002), and the theory of simulation (Zeigler et al., 2000), presented in section 4.3, two relations between formalisms can be distinguished: *embedding* and *combining* relations.

A formalism is embedded in another formalism when the first formalism is expressed in the later. A formalism combines two or more formalisms whenever all but the first formalism are subsets of the first.

Both Zeigler et al. (2000) and Vangheluwe and de Lara (2002) show that any time dependent simulation formalism can either be embedded or combined in the combined discrete-continuous formalism (DEVDESS). Multi-formalism modeling therefore imposes the following requirements for a simulation suite:

A simulation suite should provide a well interfaced DEVS & DESS simulator.

Requirement 4.3 *specification* \rightarrow *builder*: a simulation suite must at least provide model language constructs and a simulator for the combined discrete-continuous formalism (DEVDESS).

Requirement 4.4 *specification* \rightarrow *engineer*: a simulation suite must support its own extensibility with other formalisms, e.g. Petri-Net, Process-Interaction and Forrester dynamics.

4.8.2 Usability

Usability expresses the mesh between people, process and technology. Usability mainly depends on the interface between users and the decision support technology (Keen and Sol, 2005). Is it web-enabled? Can it be accessed concurrently? Is the specified layout similar to their conceptual blueprints?

Usability leads to a number of requirements for both the users and the builders of simulation models. The most important requirements for a simulation suite concerns the web.

A simulation suite should be web-enabled.

Requirement 4.5 *experimentation* \rightarrow *user*: to web-enable the complete suite. This results in web-enabled execution, output analysis, specification and storage of models, experiments and documentation.

The services of the suite should be decoupled and potentially be distributed over the internet.

Requirement 4.6 *all* \rightarrow *all*: the services of the suite should, following the late binding principle 3.4.8 and the design by contract principle 3.4.9, be decoupled and potentially be distributed over the internet.

We noted the apparent paradox of concentrating on a core while equally supporting all Us in the introduction of this thesis. We furthermore argued that this can only be achieved by orchestrating a suite of services to provide the required support. One of the main requirements for a simulation suite is thus to focus on its core functionalities. Where traditional simulation environments focus on providing animation, visualization and reporting, we argue that these subjects are not part of a simulation suite. If users are accustomed to working with spreadsheets, flowcharts or database reporting engines, why should substitutes be included in an all-in-one simulation environment?

Requirement 4.7 *all activities* → *engineer*: the suite is explicitly required to decouple the relation between a simulation core, i.e. the model and simulator, and a suite of externally provided services, e.g. reporting or animation. This results in sharing services among several suites. A simulation suite should thus be deployable in a broader decision support suite.

A simulation suite only specifies core simulation functionality.

The core notion of this research is that to provide decision support that is based on bounded rationality, multiple actors are to be supported in a decision process. To accomplish the support for a group of distributed actors in a simulation study, we come to the following requirement.

Requirement 4.8 *conceptualization and specification* → *engineer and user*: a simulation suite must support distributed model conceptualization and specification.

A simulation suite must support distributed model conceptualization and specification.

One misconception of traditional simulation environments is the tight integration between experimentation and specification environments. Environments such as Arena¹, eM-Plant² and Automod³ present one application for model execution and model specification. We see an opposite trend in software engineering; Sun's slogan on the Java programming language to *code somewhere, and run anywhere* clearly separates design from use.

Requirement 4.9 *conceptualization and specification* → *engineer*: a simulation suite must decouple applications for execution and analysis from those for conceptualization and specification.

A simulation suite must decouple applications for execution and analysis from those for conceptualization and specification.

4.8.3 Usage

Usage expresses the flexibility, adaptivity and suitability to organizational, technical, or social context. How hardware or software dependent is the technology? What can be said regarding the stability, openness and reliability?

Requirement 4.10 *all activities* → *builder and engineer*: a simulation suite should follow a standardized, well documented approach to its documentation, examples, versioning and release strategy.

Requirement 4.11 *all activities* → *engineer and builder* : a simulation suite should be provided with a set of verification tools to assess both the robustness and quality of models and of the suite itself.

A simulation suite should be provided with a set of verification tools.

¹ Arena is a trademark of Rockwell software (<http://www.arenasimulation.com>)

² eM-Plant is a trademark of Tecnomatix Technologies Ltd. (<http://www.emplant.de/>)

³ Automod is a trademark of Brooks Automation (<http://www.automod.com/>)

4.9 Summary

Having argued there is a need for decision support in the first chapter of this thesis, we presented a system of concepts that define simulation as a method of inquiry in this chapter. We introduced different modeling formalisms and described the relation between an experimental frame, a simulator and a model. We ended this chapter with a set of requirements for a simulation suite. These requirements distinguish a suite from traditional all-in-one simulation environments. A suite focuses on an orchestrated set of autonomously deployed services. The focus of a simulation suite is on providing simulation in a loosely coupled set of decision support services. We will present such a suite, named DSOL, in the following chapter of this thesis.

The focus of a simulation suite is on providing simulation in a loosely coupled set of decision support services.

5. DESIGNING A SIMULATION SUITE

A full featured simulation suite called the *Distributed Simulation Object Library*, i.e. *DSOL*, is presented in this chapter. We will start this chapter with the aim of our design: a distributed, web-enabled, service-based conceptual overview of the suite. We continue choosing an object-oriented programming language on which to base the development of our suite. Then we will give an overview of existing Java based simulation environments. We introduce the concepts underlying the design of our suite in section 5.4. The implementation of these concepts in the Java programming language is presented in sections 5.7 through 5.9.

A full featured simulation suite called the *Distributed Simulation Object Library*, i.e. *DSOL*, is presented in this chapter.

5.1 Distribution forms the core of DSOL

We present the specification of the simulation suite DSOL in this chapter. Two aspects are highlighted to emphasize the extent to which we have accomplished the N_n - N_m - N_o paradigm in the design and specification of this suite: *decoupling* and *distribution*. We argue that decoupling is *the* approach to use in the pursuit of a suite of interacting, replaceable and open services. Distribution fulfills the requirement of supporting N_n stakeholders, on N_m systems, at N_o locations. We base our definition of a service on Eckel (2000):

Definition 5.1.1 *A service is the specification of an object-oriented (sub)system that offers a cohesive set of functionality via one or more interfaces. A service is designed, implemented, and tested as a unit prior to integration into a suite of interacting services, i.e. the information system.*

Because internet technologies provide us with the ability to deploy distributed services in a modular setting, we can now put all our effort into finding and selecting appropriate services.

We argue that information system development has changed over the last decades: because internet technologies provide us with the ability to deploy distributed services in a modular setting, we can now put all our effort into finding and selecting appropriate services and orchestrating communication between them to produce a tailored information system.

The required readiness for distributed use is presented in figure 5.1. In this figure we present the concept of a simulation suite: both an experiment and a simulation model are accessible over a network. This provides users the ability to start the latest simulation environment directly from a web site, i.e. [http](http://):

`//www.simulation.tudelft.nl/dsol`, and to open an experiment file from disk which points to a third party remote web location for its simulation model (see figure 5.1). The definitions of an experimental frame, a treatment and a replication follow the system of concepts introduced in section 4.5 on page 55. Before we elaborate on the specification of DSOL, we present our reason for choosing an object-oriented language.

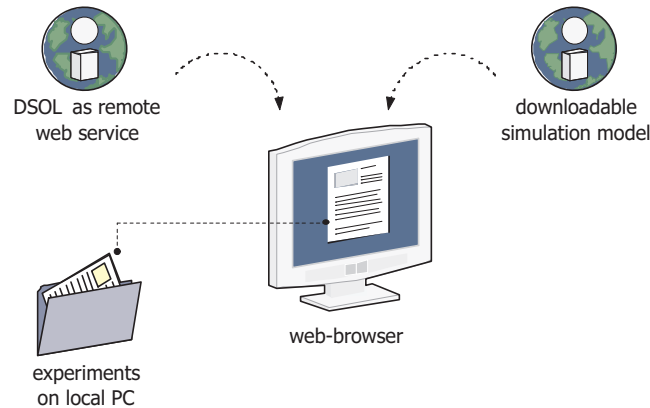


Fig. 5.1: Distributed deployment of DSOL

5.2 Choosing an object-oriented programming language

We chose to standardize on the Java programming language for the development of a simulation suite. Java is now so ubiquitous that it might appear unnecessary to comment on it. For example, Java is strict with respect to the specification of object-oriented principles. Java is furthermore inherently platform independent, and Java contains an extensive collection of libraries to support the specification of distributed, web-based applications.

We chose to standardize on the Java programming language for the development of the simulation services.

- Java allows us to express the internal structure of the simulation suite based on the well accepted and commonly used unified modeling language (UML) meta-model. By choosing the Java programming language we fulfill the requirement for a metatheoretical freedom presented on page 58.
- Java provides a strong basis for the web enabled execution of simulation models through its libraries for distributed, web-based, communication (see requirement 4.5 on page 60).

- Java allows state of the art integrated development environments (IDE), e.g. Eclipse, to be used; these IDEs support distributed collaborative model conceptualization and specification (see requirement 4.8 on page 61).
- A rich set of code verification and performance profiling services exist for the Java programming language. These services support the fulfillment of requirement 4.11 on page 61 and 4.10 on page 61.

5.3 An overview of Java based simulation environments

Numerous Java based simulation environments have been developed since the 1990s. A legitimate question is to what extent can these environments be used as a basis for the suite to be developed in this research. Kuljis and Paul (2000) present an overview of currently available Java based discrete simulation environments. We will illustrate in this section to what extent these Java based environments fulfill the requirements presented in chapter 4 and why we concluded that none of them are suitable to be used as a basis for the design presented in this chapter.

A first issue concerns the lack of a service oriented paradigm underlying most of these environments. Most environments do not provide contract based services, i.e. they do not specify classes that implement an interface and as such no interface can be presented for remote invocation, i.e. distributed deployment. Objects are not remotely accessible and the extension of these environments to a distributed simulation suite is hard if not impossible to achieve. Examples of environments which lack this service oriented paradigm are *Simjava*, *JavaSim*, and *SSJ* (Kuljis and Paul, 2000; L'Ecuyer et al., 2002).

A second issue concerns the choice of formalism. Most currently available simulation environments do not support more than one formalism. Since simulators used in these environments are targeted to execute behavior according to this specific formalism, it is difficult to redesign these environments for multi-formalism simulation. *DEVJSJAVA* is a good example of a simulation environment specifically targeted at Zeigler's DEVS formalism (Zeigler and Sarjoughian, 2005).

A third issue concerns the implementation of the process interaction formalism in Java. All currently available Java based simulation environments that support process interaction have implemented this formalism using multiple operating system threads; the approach is to use one operating system thread per process. Since the number of threads is severely limited by the operating system, simulation models specified in these environments normally cannot exceed 10^4 processes. A further complication of this issue is that the processes in the simulation model cannot be streamed¹ over a network. This makes the distributed deployment of simulation

There are several reasons why not to base our development on existing Java-based simulation services.

One, they lack a service oriented paradigm.

Two, they do not support more than one formalism.

Three, process interaction is implemented in these environments using multiple operating system threads.

¹ Streaming is the transfer of data in a continuous stream over a network (Eckel, 2000).

models impossible to achieve. Examples of multi-threaded process interaction include *SimJava*, *Silk*, *SSJ* and *JSIM* (Healy and Kilgore, 1997; Kuljis and Paul, 2000; L'Ecuyer et al., 2002). A more in-depth discussion on threads, Java and process interaction is presented in section 5.6.2.

Four, they are often published under proprietary licenses.

A final issue is that some available Java based simulation environments are published under proprietary licenses and their inner-works are often obfuscated. In the domain of software engineering, obfuscation means to make code harder to understand or read, generally for privacy or security purposes: extension of these environments is made impossible. We concluded that we would not base the design of our simulation suite on existing environments; we focused on Java as a general programming language as a starting point for our development.

5.4 Overview of the DSOL simulation suite

Before we zoom in on the Java implementation of individual services of DSOL, it is important to understand that the object-oriented systems description is not only used to conceptualize those parts of the world we want to investigate, i.e. a simulation model, but is also applied to the conceptualization of the DSOL suite; both the design of models and the suite are expressed in terms of the same systems description.

A decomposition of DSOL into horizontally layered and vertically partitioned subsystems is presented in figure 5.2. Where the connectors illustrate the decoupled autonomy between two partitions, the layered structures illustrate modular dependencies (see principle 3.2.1 on page 37) between two layers. The red blocks of figure 5.2 present external services used by the DSOL suite. The blue and yellow services make up the suite and were developed during the research process presented in this thesis. The blue services are not discussed further in this thesis. Detailed information on their design and usage can be found in Jacobs and Verbraeck (2004a). Before we present the services that make up the DSOL simulation suite, we emphasize the definition and value of the concept of a service. A service offers a cohesive set of functionalities via one or more interfaces with the aim to be integrated into a suite of independent, potentially distributed, interacting services. DSOL consists of the following services.

DSOL service

The core service provides a set of interfaces and classes for simulation. This service contains discrete and continuous formalisms, the specification of the DSOL experiment, continuous and discrete distributions, statistics and classes supporting 2-dimensional and 3-dimensional animation. This service is presented in sections 5.6 through 5.9.

The DSOL service is the core simulation service.

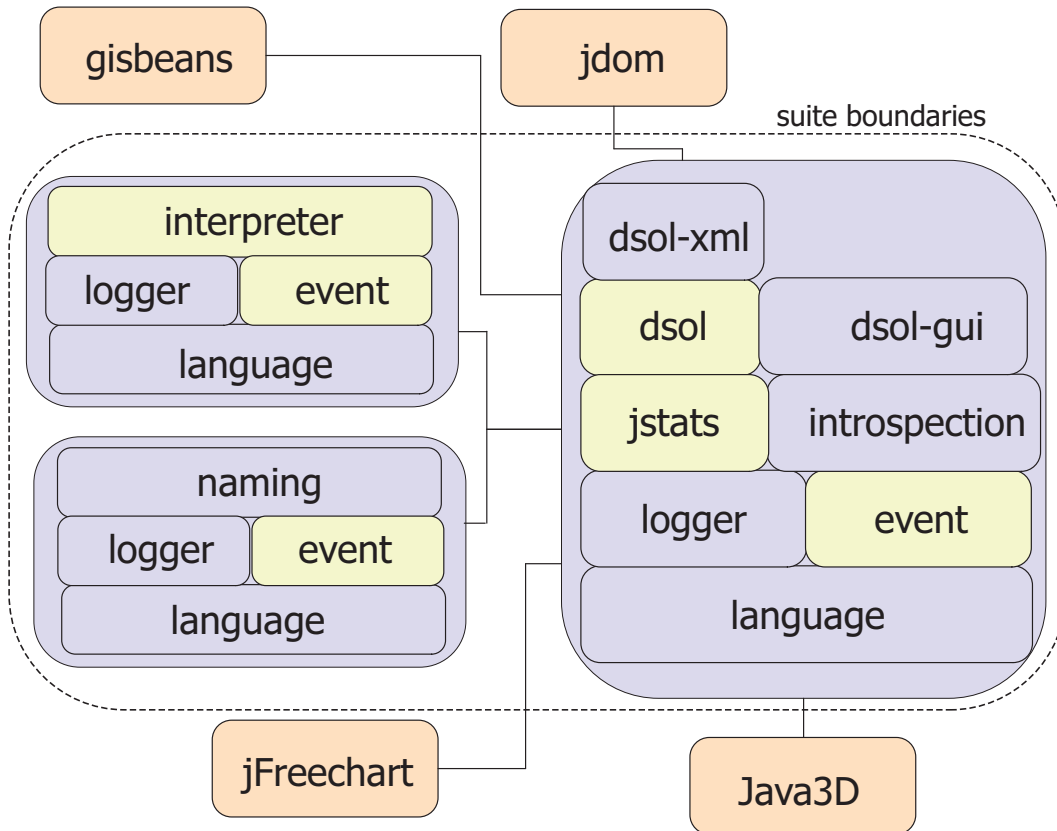


Fig. 5.2: Services making up the DSOL suite

DSOL-GUI

The collection of classes which provide a web enabled graphical user interface is presented in the DSOL-GUI service.

The collection of classes which provide a web enabled graphical user interface for DSOL is presented in the DSOL-GUI service. We emphasize the importance of designing a good user interface, and thus fulfilling requirement 4.9 on page 61 to separate environments for model development and for model execution, but we have not considered this to be part of *this* research. A reference implementation of a web enabled user interface is nevertheless presented with this service.

DSOL-XML service

The DSOL-XML service provides parsers for XML based DSOL experiments.

Over the last decade, XML has become the lingua franca² for the configuration of applications. XML is namely a platform independent, human readable language. The DSOL-XML service provides parsers for the DSOL experimental frame and as such enables users to specify an experimental frame in XML. An example is presented in appendix 9.3 on page 185. The value of this service is that it enables the specification of experiments without having knowledge of the Java programming language.

Interpreter service

The interpreter service forms the basis for the process interaction formalism in DSOL.

We argued in section 5.3 that the specification of the process interaction formalism should not be based on Java threads. To specify the process interaction formalism in Java we have developed the interpreter service: a Java virtual machine implemented in Java. This service is presented in section 5.6.2.

JStats service

The JStats service provides a set of statistical services to DSOL.

This service provides a set of continuous and discrete distribution functions and links DSOL to external mathematical and chart libraries. Jstats forms the topic of section 5.7.

Naming service

The naming service provides Yellow Page functionality to the DSOL suite.

The naming service provides Yellow Page functionality to the DSOL suite. The value of a Yellow Page service is that it provides distributed objects in the simulation suite the ability to lookup each other. The naming service provides this functionality both to simulation model objects and to those objects constituting the DSOL suite.

² Lingua Franca is a pidgin, trade language used by numerous language communities around the Mediterranean, to communicate with others whose language they did not speak (Corré, 1992).

An example of the first category of use is a supply chain simulation model in which retailers use the naming service to advertise their existence and to offer their products. An example of the second category is DSOL's animation service in which animation panels use the naming service to lookup renderables, i.e. objects that visualize simulation model objects.

The implementation of the naming service links DSOL to the Java Naming and Directory Interface (JNDI) framework. JNDI provides Java applications a unified interface to multiple naming and directory services (Sun Microsystems, 2001b). The service provided by JNDI is described in a `Context` interface. All objects in the DSOL suite use an implementation of this `Context` to lookup or bind objects. A class diagram of the implementation of this service in DSOL is presented in figure A.5 on page 184.

Introspection service

This service provides an introspection service to users; the service enables users to open a simulation model object and to introspect, i.e. to see and change, attribute values through a graphical user interface. The value of the introspection service is that it provides the ability to downdrill on simulation objects. This service aims to improve operational insight in the output of experimentation.

The introspection service enables users to open a simulation model object through a graphical user interface.

Event service

This service provides a distributed asynchronous event mechanism and thus embodies principle 3.4.7 on page 44. The value of the event service is that it enables loosely coupled relations between objects in the suite. This service is described in more detail in section 5.5.

The event service provides a distributed asynchronous event mechanism.

Logger service

The DSOL simulation suite contains a logger service which is based on Java's logging mechanism (Sun Microsystems, 2001a). The value of this service is that output, debug information, warnings, etc. produced by objects in the suite are captured and, after they are filtered and formatted, presented to subscribed listeners. DSOL's Logger service provides a set of filters and formatters to provide distributed logging.

Interdependencies in the suite

Considerable attention needs to be given to the dependencies between the different services when developing a suite. This is emphasized by the names of the services; services that are used by the simulation core, do not depend on simulation and

Tab. 5.1: Dependencies between the services in DSOL

	lang.	event	logger	naming	jstats	introspect.	interpr.	dsol	dsol-xml
language									
event	•								
logger	•	•							
naming	•	•	•						
jstats	•	•	•						
introspect.	•	•	•						
interpr.	•	•	•						
dsol	•	•	•	•	•	•	•		
dsol-xml	•	•	•	•	•	•	•	•	
dsol-gui	•	•	•	•	•	•	•	•	•

therefore do not have the *dsol* prefix in their name. Services which are add-ons to *dsol* start with the prefix *dsol*. The dependencies are illustrated in table 5.1. In the following sections we present the specification of several of these services.

5.5 Specification of distributed asynchronous communication

Objects communicate either synchronously or asynchronously in object-oriented programming languages. Synchronous communication implies that the invoker of communication requests an immediate response, while asynchronous communication implies that an answer is not immediately requested (Booch et al., 1999). It is the difference between requesting someone’s age versus requesting that one be notified when it is someone’s birthday.

The value of asynchronous communication in enabling loosely coupled relations between objects is often wrongly neglected. Asynchronous communication prevents disproportionate polling between objects and enables well tailored communication between potentially distributed objects, possibly without any prior knowledge of the class of the other object (see principles 3.4.8 and 3.4.9 on page 44). This is fundamentally required to loosely couple services, and to deploy them on the web (requirement 4.5 on page 60).

We look at the asynchronous distributed event model that is part of DSOL in this section. We show how remote event listeners are registered with event producers, and how these producers notify their subscribed listeners of state changes. We also show how subscriptions are managed by these producers.

The domains of simulation and software engineering each regrettably use the con-

We look at the asynchronous distributed event model that is part of DSOL in this section.

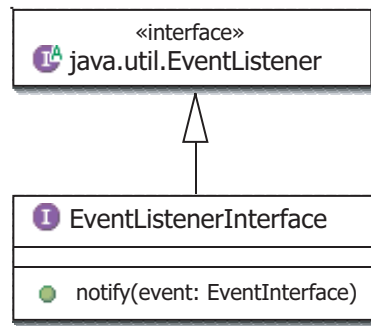


Fig. 5.3: Class diagram of EventListener

cept of event for a specific, different purpose. In the context of discrete event simulation an event is defined as a change in the state of an object at an instant. In the context of software engineering an event is a message sent between objects. To prevent confusing terminology, DSOL addresses simulation events as *SimEvents*.

In DSOL, event listeners are obliged to implement the `EventListener` interface presented in figure 5.3. The `notify` method specified in this interface ensures the required callback method for event producers on future state changes. The interface extends the `java.util.EventListener` interface which is a tagging interface that all event listener interfaces must extend (Arnold et al., 2000).

The argument passed in the `notify` method is an instance of `EventInterface` (see figure 5.4). DSOL provides two reference implementations of this `EventInterface`: a basic `Event` class and a specialized `TimedEvent`³ class containing a time stamp; the later is used for time based statistical computations. An event consists of a source, a content attribute and a type. The type, i.e. `EventType` is used to distinguish different sorts of events fired by a producer.

The `EventProducerInterface` and its reference implementation named `EventProducer` are presented in figure 5.5. In both the `addListener`⁴ and in the `removeListener` method we see that subscription is linked to a listener for a specific event type.

Two remarks must be made concerning the distributed characteristics of DSOL's event service. One, events are `Serializable`. Serialization is a way of *flattening*,

Event listeners are obliged to implement the `EventListener` interface.

Events consists of a source, a content attribute and an event type. Events are furthermore `Serializable`.

³ We reemphasize that the `TimedEvent` is not an event in the context of discrete event simulation. Such event is presented in section 5.6.1 as a `SimEvent`.

⁴ One of the arguments in the polymorph `addListener` methods is the boolean argument named `weak`; all non-distributed subscriptions are weak by default. A weak reference is a reference that does not keep the object it refers to alive; it is not counted as a reference in garbage collection (Joy et al., 2000). If the object is not also referred to elsewhere, subscription is terminated.

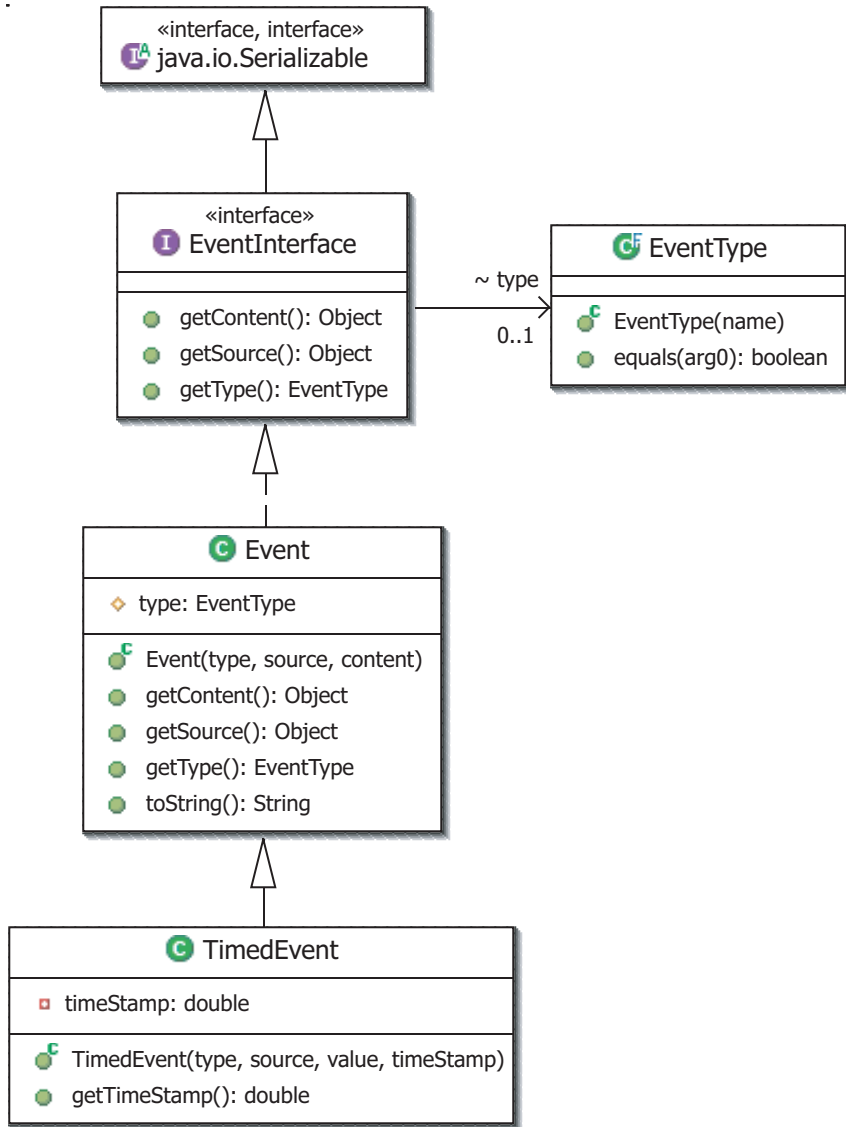


Fig. 5.4: Class diagram of Event & TimedEvent

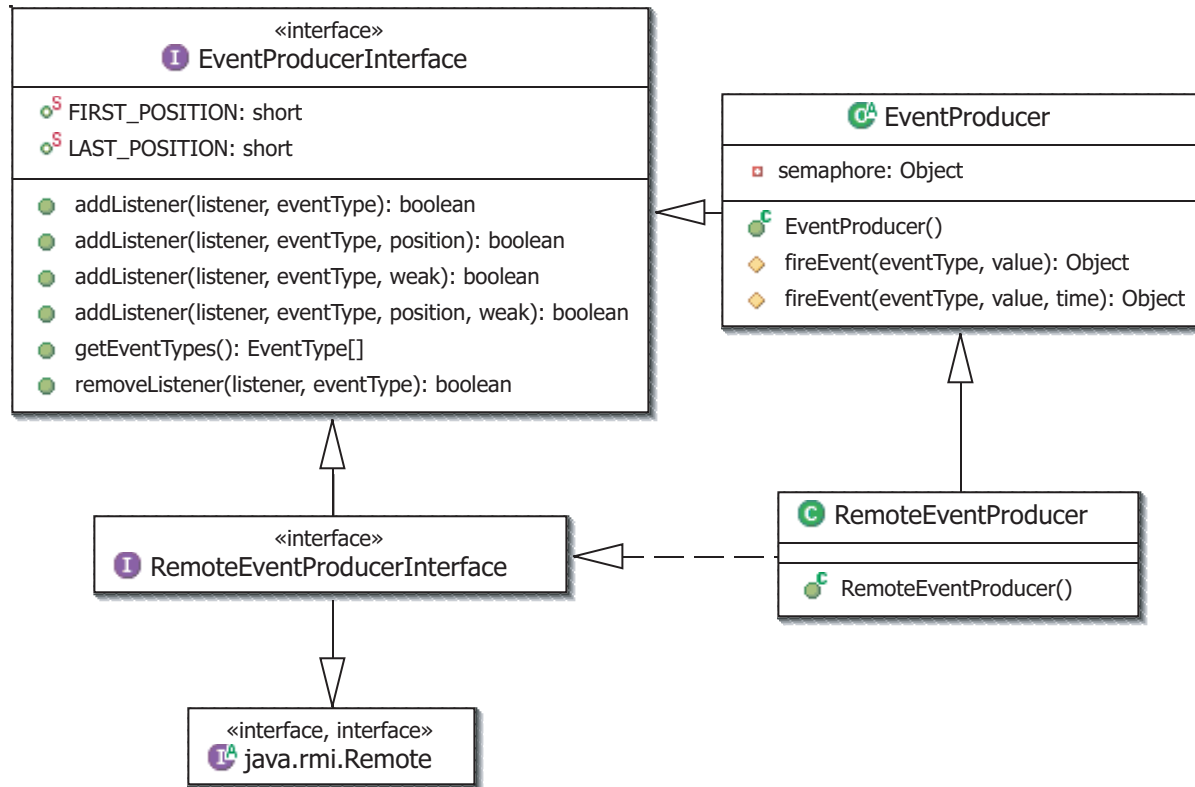


Fig. 5.5: Class diagram of EventProducer

pickling, or *freeze-drying* objects so that they can be stored on disk, and later read back and reconstituted, with all the links between the objects intact (Eckel, 2000). Serialization is required to send, or stream, events over a network. Two, both the `EventListenerInterface` and the `EventProducerInterface` can be used as remote interfaces; this is illustrated with the `RemoteEventProducer` class. This fulfills requirement 4.6 on page 60 which explicitly requires objects to be deployable on the web.

5.6 Specification of formalisms

A simulator is presented in chapter 4 as is an object, not part of the real system, that is used to generate the behavior of the simulation model. In this chapter it was stated that DSOL should provide both language constructs and simulator(s) for specific formalisms, e.g. the DEVDESS formalism.

An interface diagram of the different simulators specified in DSOL is presented in figure 5.6. The absence of classes in this figure emphasizes the decoupled, open and service oriented, i.e. interfaced, philosophy underlying DSOL.

The simulator interface specifies the formalism independent behavior of a simulator.

The `SimulatorInterface` interface presented in figure 5.6 extends the `EventProducerInterface` and specifies the formalism independent behavior of a simulator. Among its methods are those for its control, e.g. the `start`, `step` and `stop` method. A key characteristic of this interface is that it is *abstract*; it is formally unfinished and merely serves the purpose of a building block.

The extended `DEVSSimulatorInterface` and the `DESSimulatorInterface` add the attributes and behavior prescribed by the discrete event (DEVS) respectively the differential equation (DESS) formalisms. In the `DEVSSimulatorInterface` we see methods for event scheduling where the `DESSimulatorInterface` provides access to a continuous time step.

The extended DEVSSimulatorInterface and the DESSimulatorInterface add the attributes and behavior prescribed by DEVS respectively DESS formalism.

The `DEVDESSimulatorInterface` interface extends the DEVS and DESS simulator interfaces and embodies what Zeigler et al. (2000) define as a *combining* relation between the discrete and continuous formalisms. By introducing this interface we have fulfilled requirements 4.3 and 4.4 on page 60, which explicitly require the DEVDESS formalism and unrestricted to its interface.

The final extension is the `AnimatorInterface`, which introduces a realtime wall clock delay between consecutive instants of time.

To understand how specific, formalism dependent behavior is invoked, we present the reference implementation of the `SimulatorInterface` in figure 5.7. In this figure all but the final `run` method are formalism independent and thus implemented by this class. This abstract `run` method specifies the formalism dependent time advancing function and therefore has to be implemented by each subclass, i.e. simulator, individually.

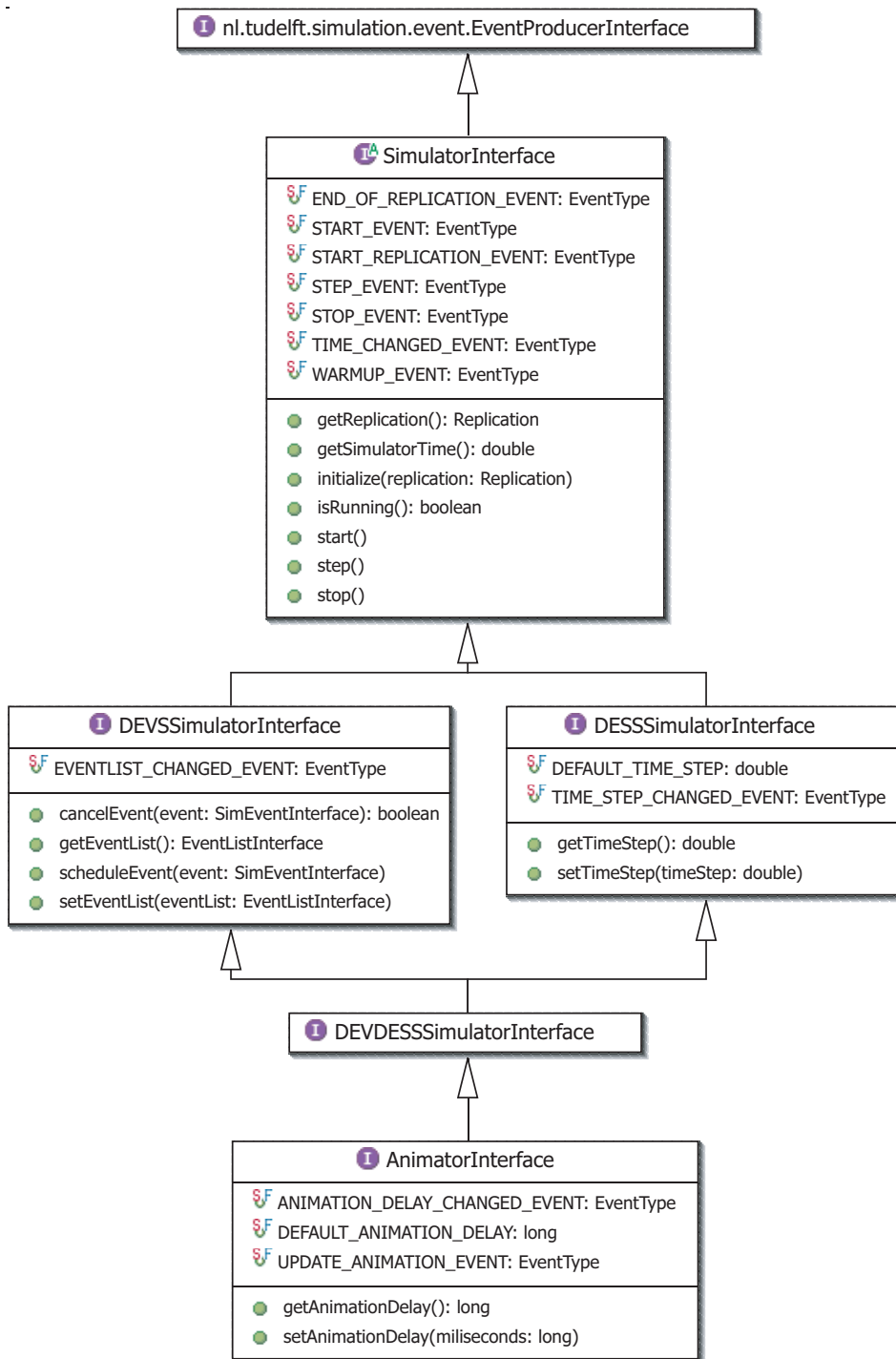


Fig. 5.6: Hierarchy between simulators

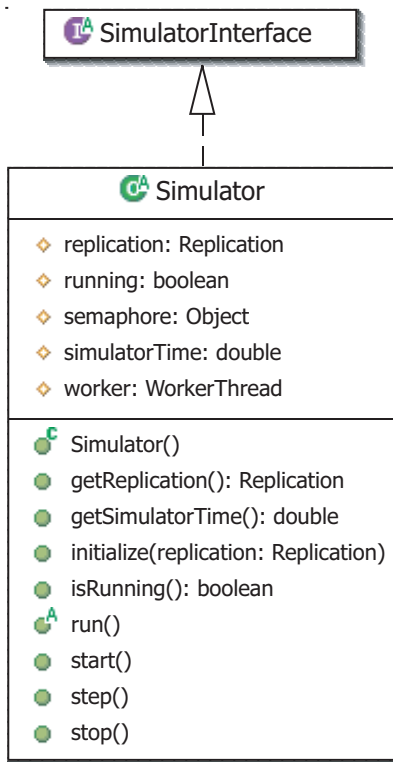


Fig. 5.7: The abstract run method in the Simulator class

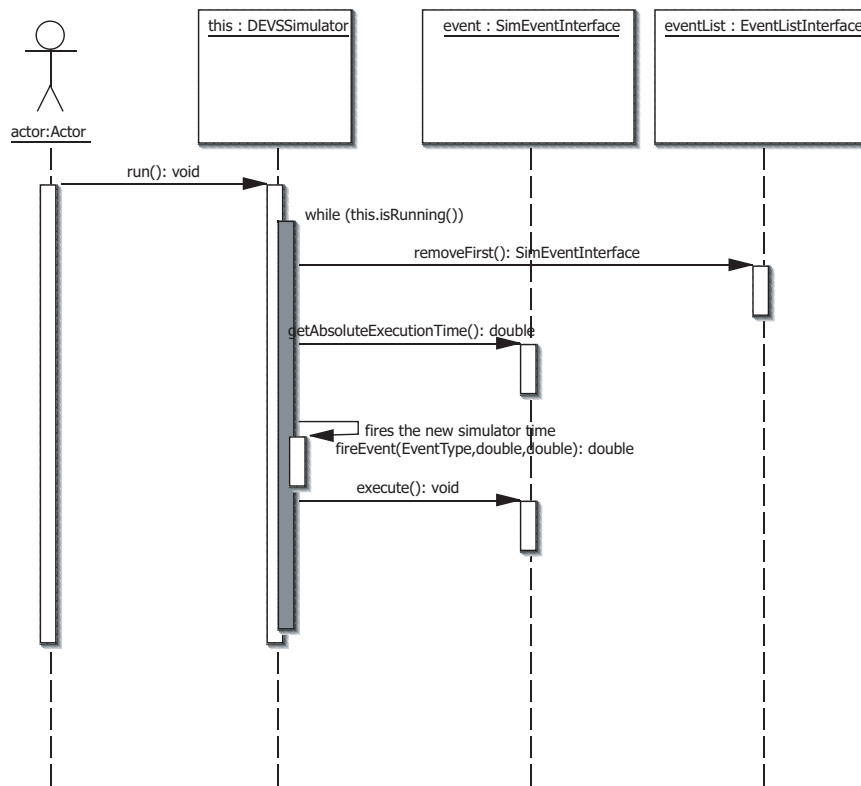


Fig. 5.8: The time advancing function of the DEVSSimulator

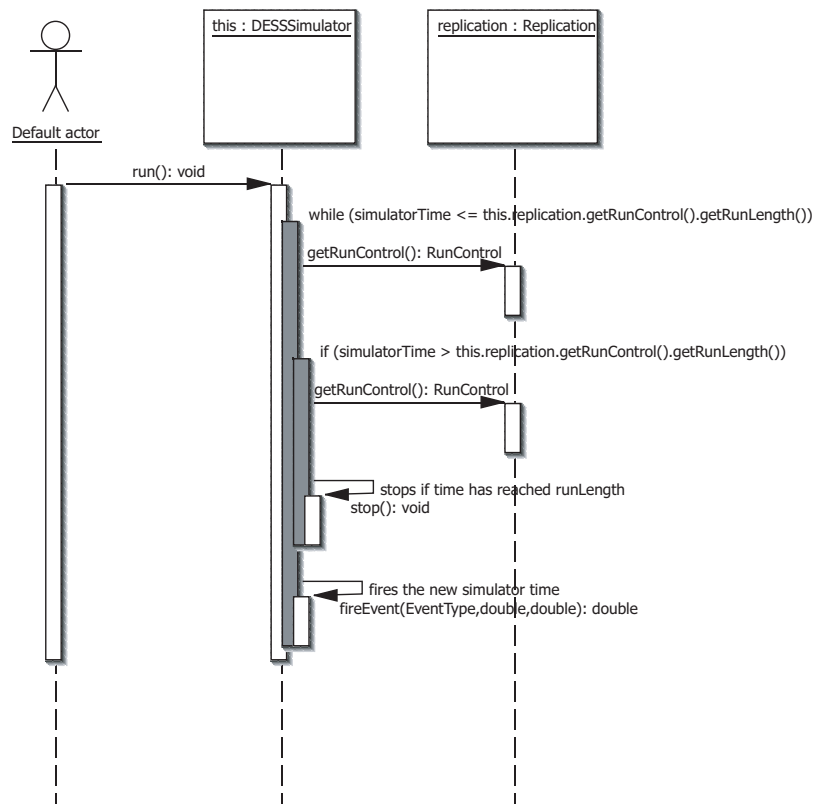


Fig. 5.9: The time advancing function of the DESSSimulator

The specifications of the discrete, continuous and combined formalisms and the sequence diagrams of their specific **run** method will now be presented. We will first discuss the **DEVSSimulator** presented in figure 5.8. As described by Balci (1988), simulator time is the time of the last executed event taken from the time sorted event list. The algorithm of the run method is presented in figure A.1 on page 177.

The time flow mechanism of the continuous **DESSSimulator** is shown in figure A.2 on page 178. A sequence diagram of this time flow mechanism (illustrated in figure 5.9) shows a continuous time advancing function. The time advancing functions of the **DEVDESSimulator**, the **Animator** and the **RealtimeClock** are presented in appendix 9.3 on page 177. It is important to recognize that they have a comparable structure to the specification of a formalism dependant **run** method.

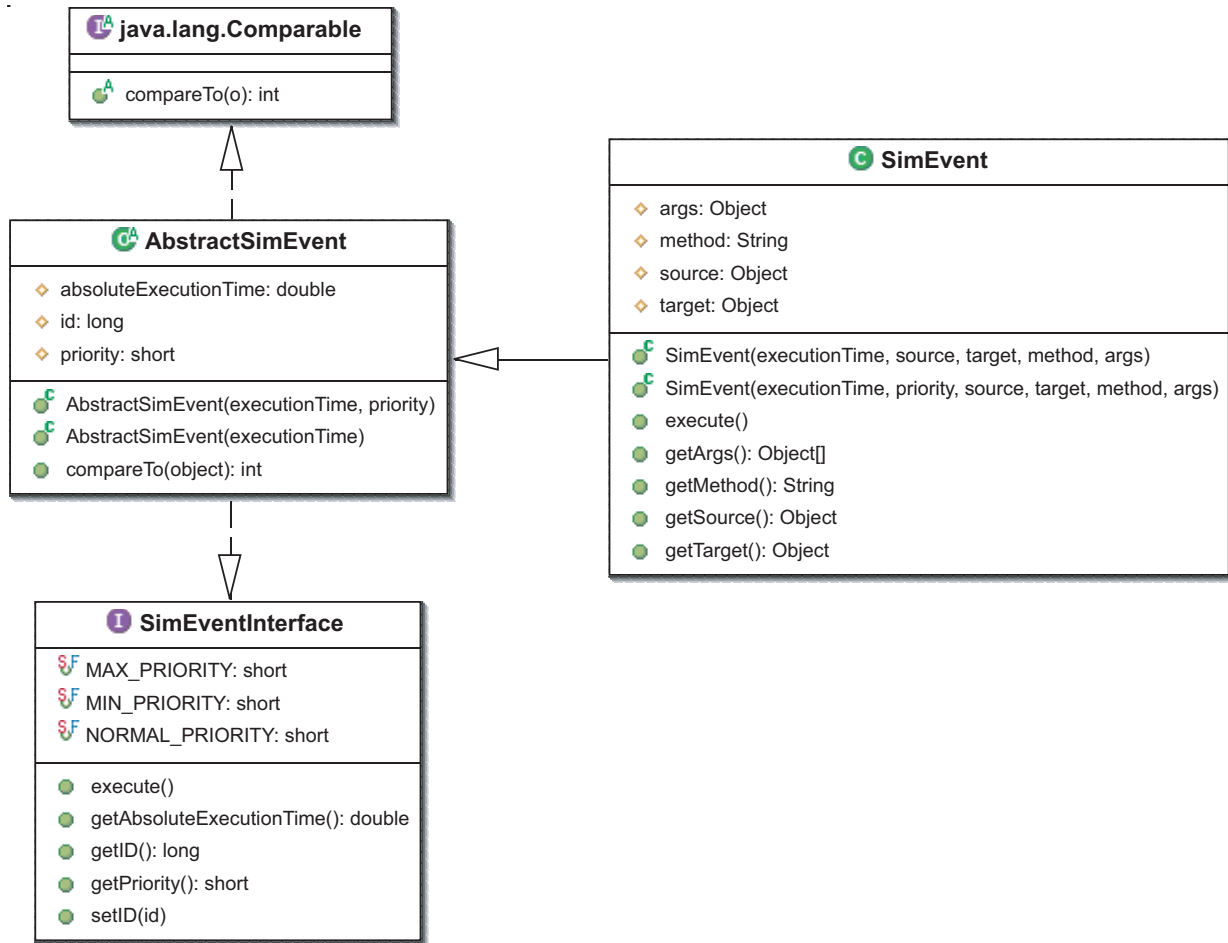


Fig. 5.10: Class diagram of discrete event formalism

5.6.1 Discrete event formalism in detail

The first formalism we describe in detail is the discrete event system specification (DEVS). The core notion concerning DSOL's implementation of this formalism is that objects used in a DEVS based simulation model do not interact using a direct method invocation, but schedule this invocation by constructing a simulation event. Such an event encompasses the scheduled execution time, a (remote) pointer to the source of the simulation event, a (remote) pointer to the object on which the method is intended to be invoked and reference to the method and the arguments with which the method is to be invoked. This mechanism is referred to as *scheduled method invocation* (Jacobs et al., 2002).

The DEVS formalism is implemented in DSOL according to a principle called scheduled method invocation.

In contrast with other formalisms, e.g. the process interaction formalism introduced in section 5.6.2, objects can directly be used in a DEVS model. The only language construct for the DEVS formalism is the `SimEvent` presented in figure 5.10. The implementation of the concept of *scheduled method invocation* is implemented in the Java programming language by applying the principle of *reflection* (see principle 3.4.10 on page 45).

To illustrate the DEVS formalism we will look at a customer that repeatedly generates orders. The following piece of code, which represents a simple M/M/1 queue, illustrates the DEVS simulation event scheduling as specified in DSOL:

```
1 package nl.tudelft.simulation.dsol.tutorial.section25;
2 public class Customer
3 {
4     private void generateOrder()
5     {
6         try
7         {
8             //We generate an order of 2 televisions
9             Order order = new Order("Television", 2.0);
10            //Now we schedule the next action at time = time + 2.0
11            SimEventInterface simEvent = new SimEvent(this.simulator
12                .getSimulatorTime() + 2.0, this, this, "generateOrder",
13                null);
14            this.simulator.scheduleEvent(simEvent);
15        } catch (Exception simRuntimeException)
16        {
17            exception.printStackTrace();
18        }
19    }
20 }
21 }
22 }
```

The DEVS formalism is presented in lines 12-16 with the construction and scheduling of a `SimEvent`. The scheduled absolute execution time of this event is the current time of the simulator plus 2⁵. The source and the target of the event point to the customer. The customer schedules the `generateOrder` method and supplies no arguments. The `DEVSSimulator` adds the event on its time sorted event list. DSOL's event list is implemented by a *Red-Black binary tree* (Wirth, 1979; Wood, 1992). In contrast to Java's reflection library, the language constructs of DSOL's DEVS implementation are all serializable which allows serialization of the `SimEvent` and thus the distribution of the `DEVSSimulator`. Again this is important to fulfill requirement 4.6 on page 60, which explicitly required objects to be deployable on the web.

DSOL's event list is implemented by a *Red-Black binary tree*.

The advantages of a DEVS implementation are its single threaded specification and the lack of requirements on objects to be used in the simulation model. In other words, where most Java based simulation languages require a specific discrete event class to be extended, we can schedule any method of any Java object in DSOL. These advantages have allowed us to fulfill requirement 4.2 on page 59.

The advantages of a DEVS implementation are its single threaded specification and the lack of requirements imposed on objects to be used.

5.6.2 Process interaction formalism in detail

Specification of the process interaction formalism in DSOL forms the topic of this section. The formal distinction between a process and an object is the fact that a process has a *control state* attribute. In its control state a process stores its reactivation point in its sequence of activities. The requirements for the `Process` class, which is presented in figure 5.11, are:

The formal distinction between a process and an object is the fact that a process has a *control state* attribute.

- the `Process` class is *abstract*. Processes as such cannot be instantiated. Classes extending `Process` are required to implement the abstract `process` method and to specify the actual sequence of activities. This is according to the principle of inheritance (principle 3.4.5 on page 42).
- the `resume` method is *public*, which expresses unlimited visibility. This is required since an object cannot `resume` itself in a suspended state.

It is far from easy to access the control state of an object in the Java programming language, to date the approach chosen by all process-oriented simulation languages implemented in Java is to circumvent access to the control state by implementing the `Process` class on top of a `Java Thread`.

It is far from easy to access the control state of an object in the Java programming language, and ...

⁵ Besides this absolute scheduling, the `DEVSSimulator` in DSOL also supports relative scheduling. The overloaded method used for this relative scheduling will now create an appropriate `SimEvent` under the hood.

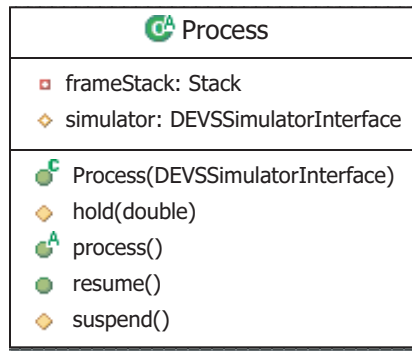


Fig. 5.11: Class diagram of process interaction formalism

Since a Java thread wraps an *operating system* thread, and operating system threads provide methods to suspend and resume themselves, processes extending a Java thread inherit these methods; the required functionality for the process interaction formalism is thus inherited through the use of Java threads. Despite the advantages of easiness and apparent correctness, this approach has some very strong disadvantages.

- Since a Java thread wraps an underlying operating system thread, Java threads are not *serializable*. Processes extending a Java thread can therefore not be streamed over a network, i.e. distributed simulation, or stored to file for model persistency.
- Since a Java thread wraps an *operating system* thread, and most operating systems can only create a limited number of threads (2000-6000), a multi-threaded specification severely limits the scalability of the simulation environment on such systems.

Given these disadvantages, we explored possible approaches to single threaded implementations. This was achieved with the introduction of *stack swapping* through a Java interpreter. A detailed discussion of this approach can be found in Jacobs and Verbraeck (2004b). To help us understand the concept of *stack swapping*, we briefly introduce the Java virtual machine specifications (Lindholm and Yellin, 1999).

... we explored possible approaches to single threaded implementations.

Whenever Java source code is compiled, a unique `.class` file is generated reflecting the compiled byte code for the particular class. Although the format of this file may seem unreadable, it consists of integers, shorts, utf-8 characters, etc.

A `.class` file starts with several constants defining the name of the class, its superclass and the set of interfaces it implements. Then the *constantpool* is specified. The constants in this pool are used by the fields and methods described in

the `.class` file and merely serve the purpose of preventing unnecessary bytes in the file; a pointer to the 3rd constant in a constantpool of 255 positions occupies only 1 byte, where duplicating a complete string will most certainly occupy more memory.

All fields and methods of a class are described by their signature, i.e. name, return type and possible parameters. The body of a method is specified as a list of sequential *assembly operations*. The Java virtual machine specification (Lindholm and Yellin, 1999) introduces almost 200 operations. Whenever a method is invoked, a Java virtual machine simply sequentially executes this list of operations.

In software terms, the invocation of a method is called a *frame* and a *stack* represents a last-in-first-out (LIFO) stack of objects. A Java virtual machine uses two types of stacks to execute the invocation of a method. One is a *frame stack*, exclusive to a thread. The other is an *operand stack* which represents a stack for operations and is exclusive to each frame, or method invocation. As will become obvious from the following example, another object used in the execution of a frame is the pool of *localvariables*. This pool holds all local variables of a method, including the method-parameter values.

The execution of byte code can perhaps best be described by a simple example. Consider a class defining two mathematical operations `square` and its more general `pow`.

If `square(4.0)` is invoked, the main thread of execution creates a new *frame* for this invocation and pushes this frame on top of its *frame stack*. Then it starts the execution of this frame resulting in the execution of the following 4 assembly operations:

```
DLOAD_0      //stack.push(localvariable(0))
ICONST_2     //stack.push(constantpool(1))
INVOKESTATIC //invoke pow(stack.pop(),stack.pop())
DRETURN      //return stack.pop()
```

Based on the source-code presented in this example, it is not difficult to understand that the `INVOKESTATIC` operation invokes the more general `pow(a,2)` method. A new frame is thus created and pushed on top of the previously created frame representing the `square` invocation. The thread first executes this newly pushed frame before it resumes the `DRETURN` operation. The operations of the `pow` frame are:

```
ILOAD_2      //stack.push(localvariable(2))
ICONST_1     //stack.push(constantpool(1))
IF_ICMPLE    //if(..<..)
DLOAD_0      //stack.push(localvariable(0))
ILOAD_2      //stack.push(localvariable(2))
```

```

ICONST_1      //stack.push(constantpool(1))
ISUB          //stack.push(stack.pop()-stack.pop())
INVOKESTATIC //invoke pow(stack.pop(),stack.pop())
DLOAD_0      //stack.push(localvariable(0))
DMUL         //stack.push(stack.pop()*stack.pop())
DRETURN      //return stack.pop()
DLOAD_0      //stack.push(localvariable(0))
DRETURN      //return stack.pop()

```

Since the `pow` method is a recursive method, the number of frames created for its invocation is equal to `b-1`. Whenever a frame returns a value, this value is pushed on its *parent* frame. Then the frame is removed from the framestack and the execution of the parent frame resumes.

The approach introduced as stack swapping works as follows: assume a `Process` is paused with a `hold` statement. The simulator thread now:

- pops its framestack up to the point where the `process` method of the `Process` was invoked. Since the `process` method is `void`, no return value is pushed to the parent frame.
- stores this popped part of its framestack as the *control state* of the `Process`.
- stores the index of the last executed operation as the *reactivation point* of this control state.
- continues the execution of its framestack which results in executing the next scheduled simulation event. The `suspend` is now successfully accomplished.

Whenever the simulator thread resumes a `Process` it pushes the *control state* of the process on its framestack and resumes its new top frame on the specified *reactivation point*. In contrast to the *multi-threaded* and *method splitting* approaches presented in Jacobs and Verbraeck (2004b), *stack swapping* has no constraints; it is therefore the preferred approach for the specification of the *process interaction* formalism. One remaining problem is that the Java programming language does not provide any language constructs to access either the framestack of a thread, or its local variables.

The approach chosen in DSOL was to develop such language constructs. These constructs, i.e. library, can best be seen as a virtual machine implemented in Java. A stack, a constant pool and the set of 200 operations were implemented to accomplish the specification of this library. They make up the interpreter service in DSOL. To prevent any overhead resulting from the use of this service, a tight invocation scheme is used in DSOL: only invocation of pausable methods is interpreted. All other invocations are not interpreted, but directly invoked through

reflection. With the interpreter service, we have successfully implemented the process interaction formalism at a small expense of interpreting classes which extend `Process`.

The approach for the implementation of the process interaction formalism in the DSOL simulation suite now became:

- a `Process` class is specified as introduced in the class diagram in figure 5.11. This class has one attribute called `framestack` which type is `java.util.Stack`.
- the constructor of the `Process` schedules on the simulator the *interpretation* of the `process` method to be invoked at `time=0.0`. The interpretation is thus scheduled!
- Starting at `time=0.0`, the `process` is interpreted sequentially and hierarchically.

The process interaction example presented in this section is drawn from the early years of Simula '67 (Birtwistle, 1979). In this example we consider boats entering a port. Whenever a boat enters the port it first claims 1 jetty. After a jetty is assigned the boat requests for 2 tugs to help it to dock its vessel. Docking takes 2 minutes after which the boat releases the tugs. Now the boat unloads its cargo which takes 14 minutes. To leave the port the boat requests 1 tug for 2 minutes. Once the boat has left the jetty, both the tug and the jetty are released. We merely present the the specification of the `Boat` class in this chapter. The `Port` class and `Model` class are presented in appendix 9.3 on page 181.

```
1 package nl.tudelft.simulation.dsol.tutorial.section45;
2
3 public class Boat extends Process
4     implements ResourceRequestorInterface
5 {
6     public void process() throws RemoteException
7     {
8         double startTime = this.simulator.getSimulatorTime();
9
10        //We request and seize one jetty
11        this.port.getJetties().requestCapacity(1.0, this);
12        this.suspend();
13
14        //Now we request and seize 2 tugs
15        this.port.getTugs().requestCapacity(2.0, this);
16        this.suspend();
```

```
17
18 //Now we dock which takes 2 minutes
19 this.hold(2.0);

20
21 //We may now release two tugs
22 this.port.getTugs().releaseCapacity(2.0);
23
24 //Now we unload
25 this.hold(14);
26
27 //Now we claim a tug again
28 this.port.getTugs().requestCapacity(1.0, this);
29 this.suspend();
30
31 //We may leave now
32 this.hold(2.0);
33
34 //We release both the jetty and the tug
35 this.port.getTugs().releaseCapacity(1.0);
36 this.port.getJetties().releaseCapacity(1.0);
37 }
38
39 public void receiveRequestedResource(final double capacity,
40     final Resource resource)
41 {
42     this.resume();
43 }
44 }
```

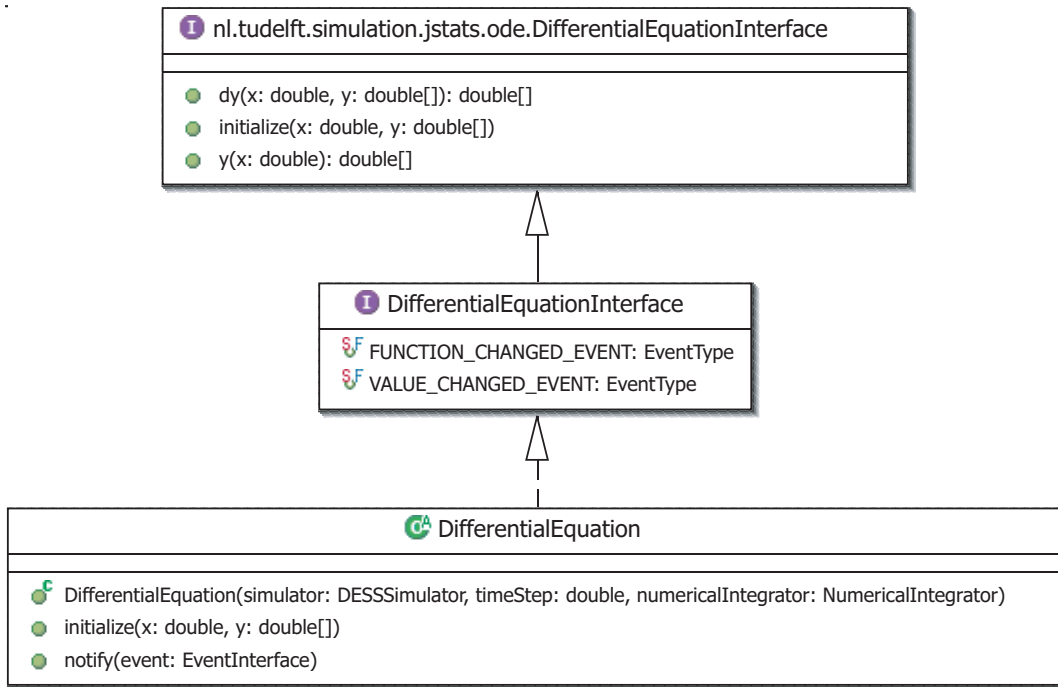


Fig. 5.12: Class diagram of differential equation formalism

5.6.3 Differential equation formalism in detail

The last formalism discussed in this chapter is the *Differential Equation System Specification (DESS)*. In this formalism differential equations are solved using numerical integration. Numerical integration is the approximate computation of an integral using numerical techniques (Faires et al., 2002). The language constructs for the DESS formalism are presented in figure 5.12.

As illustrated in figure 5.12 a differential equation must extend DSOL's abstract `DifferentialEquation` class. These subclasses thus specify the `dy()` method in which the value `y` is specified as an array of doubles. DSOL thus expects any n^{th} order ordinary differential equation to be rewritten as an array of first order differential equations. Faires et al. (2002) present a number of integration techniques for ordinary differential equations (ODE). They differ in approach, error and efficiency. As illustrated in chapter 6, different models require different integrators.

As an example of the DESS formalism we consider the well known distance function $x(t) = 0.5at^2 + v_0(t) + x_0$, with $a = 0.5$, $x_0 = 0$ and $x_0 = 0$.

The last formalism discussed in this chapter is the *Differential Equation System Specification (DESS)*.

A differential equation must extend DSOL's abstract `DifferentialEquation` class.

```

31 public class Distance extends DifferentialEquation
32 {
39     public Distance(double timeStep, short integrator)
40     {
41         super(timeStep, integrator);
42     }
48     public double[] dy(double x, double[] y)
49     {
50         return new double[]{0.5, //dspeed = a
51                               y[0]}; //ddistance = speed
52     }
59     public static void main(String[] args)
60     {
61         Distance distance = new Distance(0.0001,
62         NumericalIntegrator.EULER);
63         distance.initialize(0,new double[]{0,0});
64     }
65 }

```

All numerical integrators provided in DSOL are presented in table 5.2.

In lines 48-52 the distance is specified as a set of first order differential equations. Lines 61-62 construct the differential equation using Euler's numerical integration algorithm and a time step of 0.0001. All numerical integrators provided in DSOL are presented in table 5.2. The implementations are based on the algorithms described in Weisstein (1999). The numerical values in this table refer to the computational efficiency of the integrator with respect to the *Euler* integrator ($\equiv 1.000$). These measurements are based on the given equation and time step and are conducted on a 1Ghz Pentium III, 512MB RAM system with DSOL 1.6 on JRE 1.4.2.

5.7 Specification of statistical distribution functions

In this section we discuss how to specify the probabilistic nature of the natural phenomena under investigation in a simulation study. We start with the concept of randomness and continue with several continuous and discrete distributions.

5.7.1 Pseudo random number generators in detail

The generation of pseudo random numbers is considered to be a requirement for any general purpose simulation environment (see requirement 4.1 on page 59). As Knuth (1998) explains, randomness does not refer to an individual number but to

Tab. 5.2: Numerical integrators implemented in DSOL

Numerical integrator	Computational efficiency
Euler	1.000
Heun	0.434
Runge Kutta 3	0.289
Runge Kutta 4	0.212
Gill	0.171
Milne	0.089
Adams	0.088
Runge Kutta-Fehlberg	0.044
Runge Kutta-Cash Carp	0.043

a sequence of numbers. A uniform distribution on a finite set of numbers is one in which each possible number is equally probable.

The next question concerns the use of random numbers. Random numbers are used in the specification of probabilistic distributions representing the real system in such a model. Before we explain how the uniform distribution is used as a basis for several continuous and discrete distributions, we illustrate DSOL’s pseudo random number generators in figure 5.13.

First a `StreamInterface` is presented which specifies a contract to be observed by any generator. In using this interface we underline once more the openness of DSOL; potential usage of competing implementations is inheritably supported which is in line with the design by contract principle on 44.

The first generator we discuss is the `Java2Random` class. This class extends Java’s reference implementation of a random number generator and enables its use in DSOL. As specified by Arnold et al. (2000) this is a 48-bit linear congruential random number generator with $X_{n+1} = (aX_n + c) \bmod(m)$. In this equation a and c are constants and m is a 48-bit modulus value (Knuth, 1998).

The disadvantages of this linear congruential method, which was introduced in Lehmer (1951), are its relatively short period, and its questionable randomness (L’Ecuyer, 1997). The period of Java’s 48-bit reference implementation is $\approx 10^{14}$. Deng and Xu (2003) presented a category of high-dimensional, efficient, long-cycle and portable uniform random number generators based on L’Ecuyer et al. (1993); L’Ecuyer and Simard (1999). The `DX-120` class in figure 5.13 specifies this generator, the period of which is $\approx 10^{1120}$.

The last generator presented in figure 5.13 is the Mersenne Twister. This generator, presented by Matsumoto and Nishimura (1998), has an astronomical period of $\approx 10^{6000}$. The increased period of the latter two generators comes at the expense of computational efficiency; the `nextDouble()` operation performs 18 percent less

DSOL provides three pseudo random number generators: the 48-bit LCG, a DX-120 generator and a MersenneTwister generator.

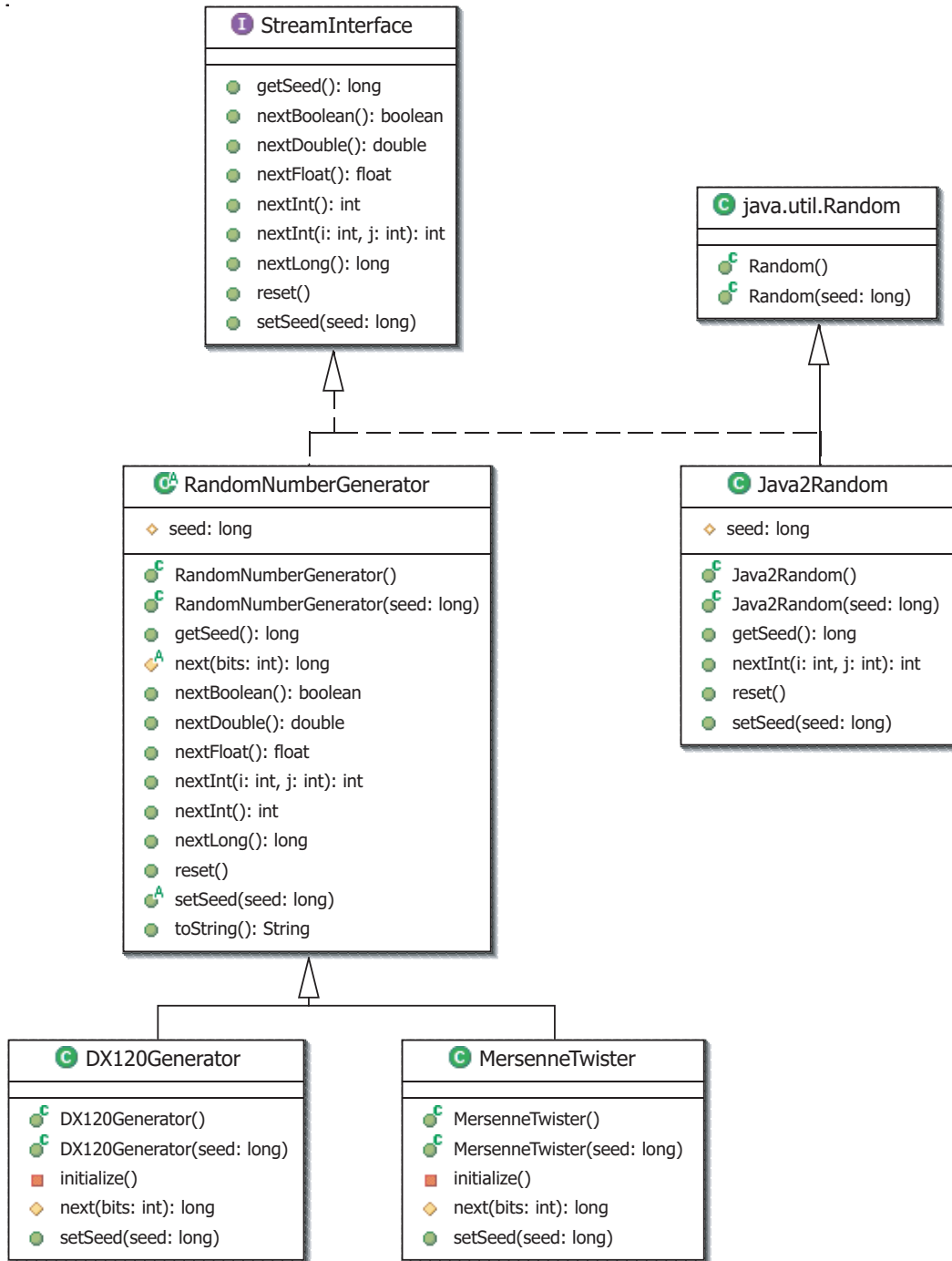


Fig. 5.13: Random number streams

efficient for the DX120 and 32 percent less for the Mersenne Twister compared to the 48-bit linear congruential method. These measurements were performed on a 1Ghz Pentium III, 512MB RAM system with DSOL 1.6 on JRE 1.4.2.

5.7.2 Statistical distributions in detail

Law and Kelton (2000) present particular algorithms for generating random variates for several commonly occurring continuous and discrete distributions. As explained, these variates form the basis for all probabilistic simulation processes. Law and Kelton (2000) state that there are often several algorithms that can be used for generating variates from a given distribution. In these cases they selected a satisfying one with respect to the tradeoff between computational efficiency and easiness of understanding.

A class diagram of the statistical distributions in DSOL is presented in figure 5.14. The root class in this diagram defines the abstract `Dist` class with its `stream` attribute. The `Dist` class is extended by two abstract classes: the `DistContinuous` and the `DistDiscrete` class. Where the continuous distribution draws continuous random variates, the discrete distribution returns discrete random variates. Two specific distributions are illustrated in figure 5.14: the exponential and Poisson distribution. All distributions currently provided by the DSOL suite are presented in table 5.3.

A class diagram of the statistical distributions in DSOL is presented in figure 5.14.

Tab. 5.3: Random variates in DSOL based on Law and Kelton (2000)

Continuous distributions	Discrete distributions
Beta	Bernoulli
Constant	Binomial
Erlang	Empirical
Exponential	Constant
Gamma	Uniform
Normal	Geometric
LogNormal	Negative Binomial
Pearson 5	Poisson
Pearson 6	
Triangular	
Uniform	
Weibull	
Empirical	

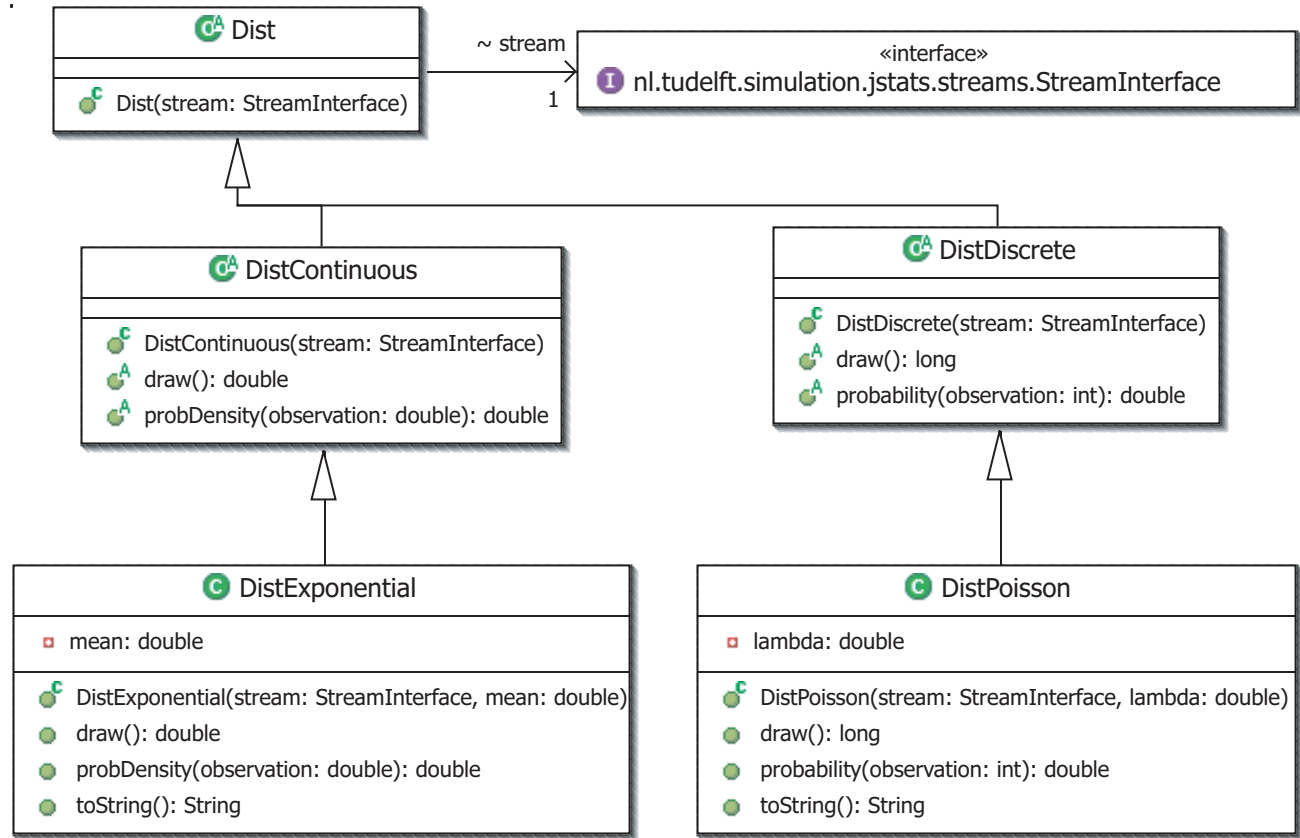


Fig. 5.14: Statistical distributions

5.8 Specification of output statistics

The statistical objects in DSOL is presented in figure 5.15; they include a **Counter**, a time independent **Tally** and a time dependent **Persistent**. Algorithms for the estimation of variance, standard deviation and confidence intervals come from Law and Kelton (2000).

More important is the notion that the design of statistical objects is based on the distributed asynchronous event model presented in section 5.5. This results in the following characteristics of the statistical output in DSOL:

- there is a clear decoupling between a statistics object and its event source, i.e. producer. A model object does not have to have to know that statistics are collected on its performance, but they need to be instances of the **EventProducer** class in order to asynchronously fire relevant events.
- statistics objects use a filter to filter incoming events. Implementations of this **FilterInterface** include a modulus filter which accepts every n^{th} event, and a max filter which accepts the first n events. Filters can be inverted and combined.
- statistics objects and their event sources can be distributed over a network. This is the result of the serializable events and the **RemoteEventProducer** interface (see section 5.5).
- statistics objects are event producers and fire changes in their values, e.g. mean and variance. These events are used by decoupled, potentially distributed charts of the DSOL-GUI service. The open source, external JFreechart⁶ library is used for the creation of these charts. This illustrates the openness of the suite, the willingness to use external services and as such (partly) fulfills requirement 4.7 on page 61 which states that only a core simulation suite should be designed.

The statistical output objects in DSOL are presented in figure 5.15.

The design of these objects is based on the distributed asynchronous event model.

⁶ JFreeChart is a free Java class library for generating charts (<http://http://www.jfree.org/jfreechart/>)

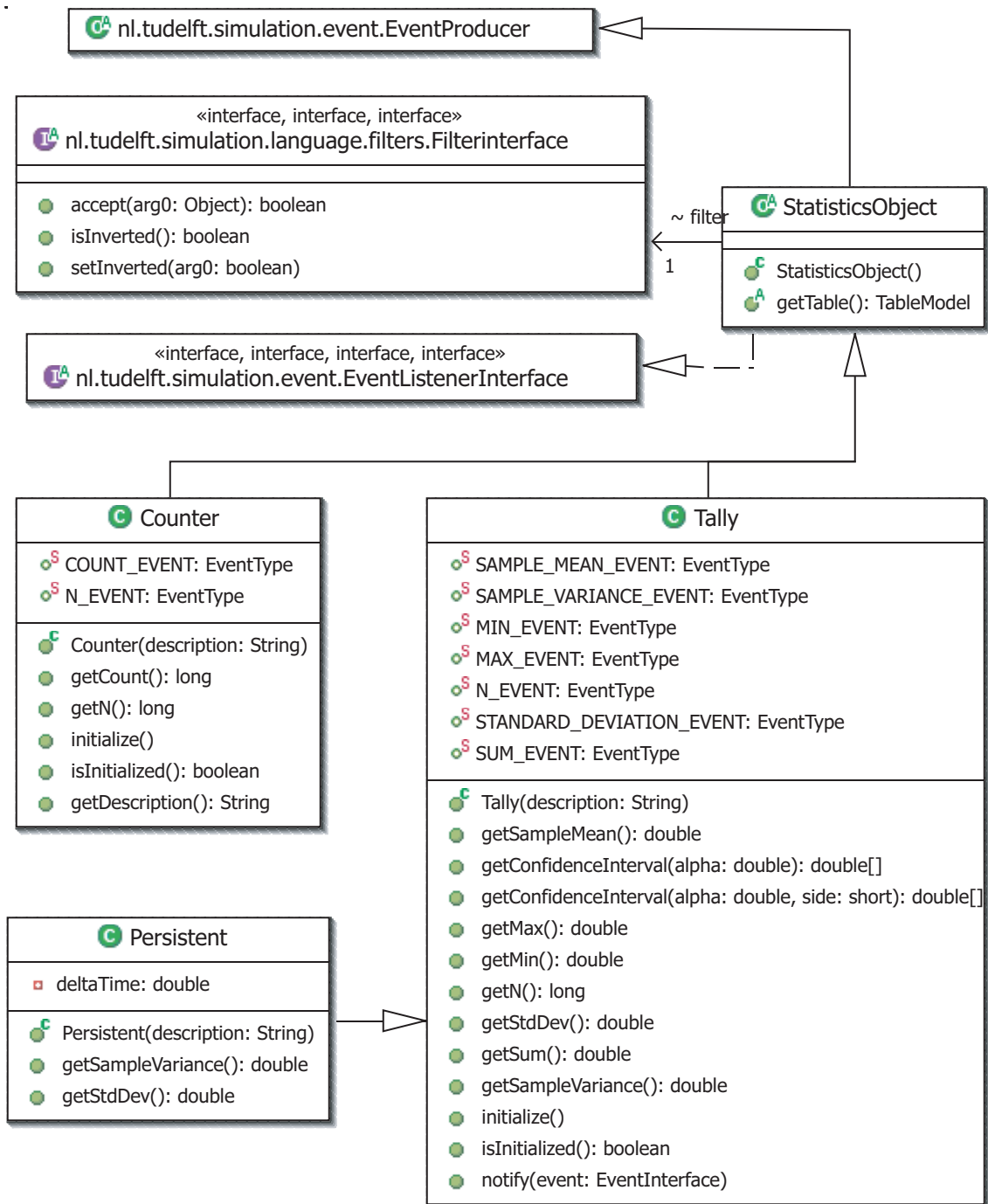


Fig. 5.15: Output statistics

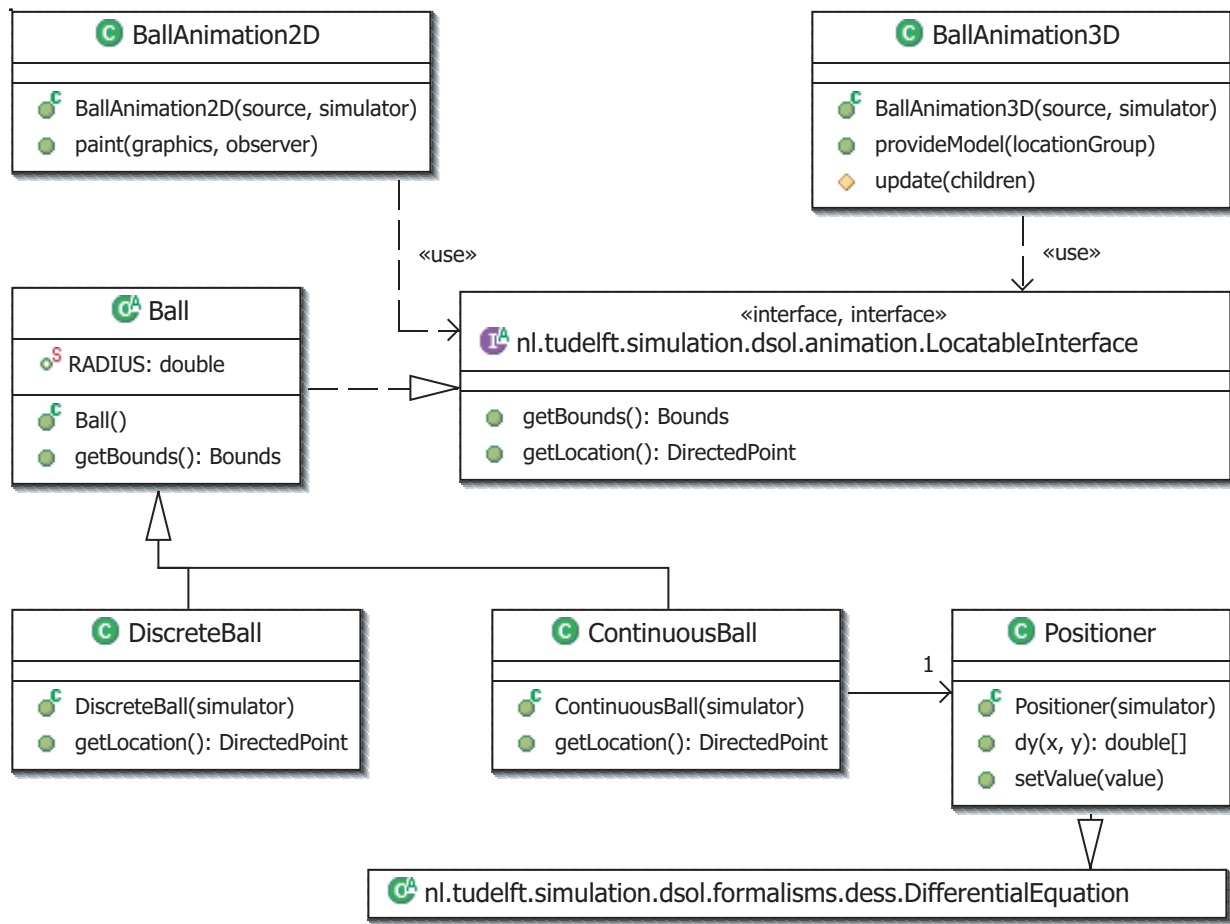


Fig. 5.16: Animation in DSOL: the conceptual design

5.9 Specification of animation

We present DSOL’s animation features in this section. The reason for introducing this service is that, besides the contribution good animation brings to usability, especially ease of understanding, this service serves as an excellent example of the decoupling and distributed principles described in the introduction of this chapter. The particular example of this section furthermore illustrates the concept of multi-formalism simulation.

The class diagram of an animation example is presented in figure 5.16; in this figure balls move around. The abstract `Ball` class implements the `Locatable` interface which describes the physical location of an object and specifies its `getBounds` method. The bounds of an object define a convex, closed volume that is used for various intersection and culling operations. The prescribed `getLocation` is for-

The class diagram of an animation example is presented in figure 5.16.

A ball can be animated by multiple instances of both the 2D and the 3D animation classes.

malism dependent and therefore implemented by the `DiscreteBall` respectively the `ContinuousBall`. A suitable `DEVDESSSimulator` enables the specification of a simulation model in which continuous and discrete balls move around; hence the concept of multi-formalism simulation. The following classes and relations are worth discussing.

- A `Positioner` class is introduced to compute the time dependent position of a continuous moving ball. Although it would be more elegant if the `ContinuousBall` would extend the `DifferentialEquation` directly, this is impossible: the Java programming language does not support multiple inheritance.
- The absence of a relation between model components, e.g. a ball, and animation components, e.g. an instance of the `BallAnimation` class clearly shows that animation is DSOL is based on a pull mechanism. If no animation component is instantiated, or if the current simulator is not an instance of the `Animator` class, or if no graphical user interface is openend, animation does not occur. Animation is thus completely separated from simulation.

Decoupling is achieved by the absence of a relation between the `BallAnimation2D`, the `BallAnimation3D` and the `Ball` classes. Animation objects merely expect a `Locatable` source, which provides required information to present a visual representation on the correct location. As a consequence, a ball can be animated by multiple instances of both the 2D and the 3D animation classes and an animation class can animate both continuous, discrete and any other `Locatable` object (see figure 5.17). The clear distinction between a `Locatable` model object and one or more animation objects further enables their distributed deployment; hence the concept of distribution. In such case multiple distributed animation screens represent the same model. A screen shot is presented as an example in figure 5.17.

The clear distinction between a `Locatable` model object and one or more animation objects further enables their distributed deployment.

5.10 Summary

We introduced DSOL in this chapter: a set of loosely coupled, substitutable, open and interacting services aimed at providing effective decision support through simulation as the method of inquiry. DSOL forms the the *piece de resistance* of this research; in it we present the core of our research and as such we presented in this chapter our contribution to the domain of systems engineering. In chapter 4 we presented a number of requirements. Our conclusion that we have fulfilled these requirements is presented throughout this chapter and summarized in table 5.4.

At the time of writing this thesis (2005), version 2.0.0 of DSOL was released. This release contained over 700 classes and more than 90,000 lines of documented

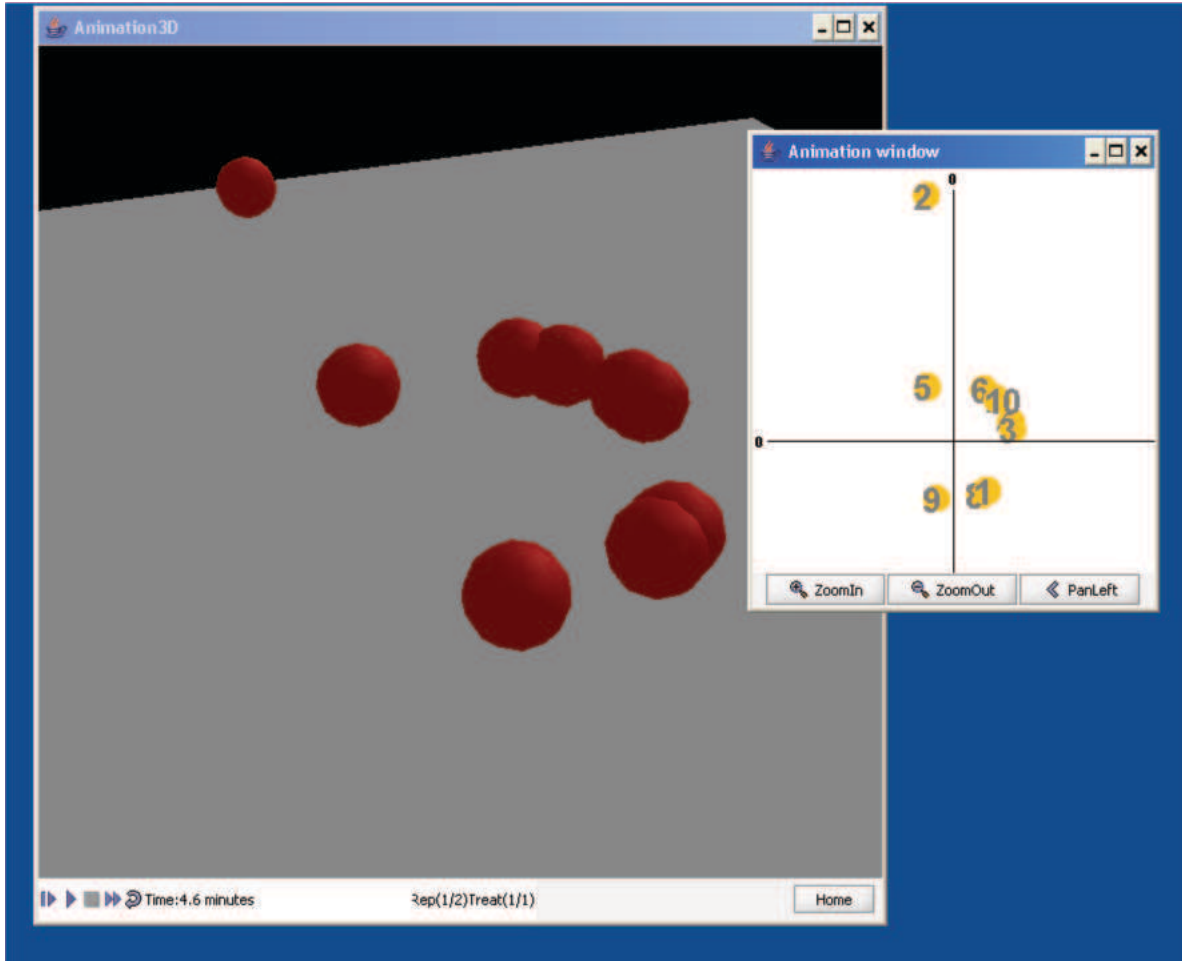


Fig. 5.17: 2D and 3D animation of continuous and discrete event moving balls

Tab. 5.4: Summary of the fulfillment of the design requirements for DSOL

Requirement	Fulfilled	Page
Requirement 4.1 on page 59 (<i>usefulness</i>)	✓✓	on page 88
Requirement 4.2 on page 59 (<i>usefulness</i>)	✓✓	on page 81
Requirement 4.3 on page 60 (<i>usefulness</i>)	✓✓	on page 74
Requirement 4.4 on page 60 (<i>usefulness</i>)	✓✓	on page 74
Requirement 4.5 on page 60 (<i>usability</i>)	✓✓	on page 64
Requirement 4.6 on page 60 (<i>usability</i>)	✓✓	on page 74
Requirement 4.7 on page 61 (<i>usability</i>)	✓✓	on page 93
Requirement 4.8 on page 61 (<i>usability</i>)	✓✓	on page 65
Requirement 4.9 on page 61 (<i>usability</i>)	✓✓	on page 68
Requirement 4.10 on page 61 (<i>usage</i>)	✓✓	on page 65
Requirement 4.11 on page 61 (<i>usage</i>)	✓✓	on page 65

Java source code. This chapter is therefore the result of an insuperable tradeoff between readability and the completeness of describing all architectural concepts. A tutorial illustrating the ins and outs of using the suite is not included in this thesis, however such a tutorial can be found in Jacobs and Verbraeck (2004a). We address the verification and validity of DSOL in the following chapters. We furthermore address DSOLs future and as such aim to firmly root it in the simulation community.

6. VERIFICATION AND TESTING OF DSOL

The verification of DSOL forms the central theme of this chapter. Roache (1998); Balci (1995); Sol (1982); Mitroff et al. (1974) define verification as the process of determining that a system implementation accurately represents the developer's conceptual description of the system. In the following sections we provide scientific evidence for the verification of DSOL based on a substantive perspective, which implies that DSOL is tested with respect to the correctness and completeness of its algorithms. Testing and analyzing the simulation suite is given in section 6.2.

The verification of DSOL forms the central theme of this chapter.

6.1 *Expert verification through the SNE comparisons*

Simulation News Europe (SNE) features a series on comparisons of simulation software. Special features of modeling and experimentation are compared based on simple, easily comprehensible models. The effectiveness of support of DSOL is more complex, real-life environments form the topic of chapters 7 and 8.

Simulation News Europe features a series on comparisons to verify the implementation of...

The features compared are for instance modeling technique, event handling, numerical integration, steady-state calculation, distribution fitting, parameter sweep, output analysis, animation, complex logic strategies, submodels, macros and statistical features (Breitenecker, 2004). For this research the implementation of these tests served a number of goals.

...modeling technique, event handling, numerical integration, steady-state calculation, etc.

- Since these tests have been implemented in a number of simulation software packages, the specification in DSOL provided a strong opportunity to verify the mathematical algorithms and output of DSOL.
- These tests provided a strong opportunity for the validation of the completeness of the suite with respect to its features and its usefulness. This validation is strengthened by SNE's scientifically objective specification of evaluation criteria.
- These tests served as attractive student assignments and their specification serves as good tutorial material for the development community.
- The specification of these tests provided us with the opportunity to reach the simulation community, given SNE's willingness to publish them.

At the moment of writing Breitenecker (2004) gives the following set of tests.

1. *Lithium-cluster dynamics under electron bombardment*: this comparison deals with a stiff system of the 3rd order. This comparison tests features for integration of stiff systems, for parameter variation and for steady state calculation.
2. *Flexible assembly system*: this comparison is targeted at discrete event simulation languages and compares features for submodel structures, control strategies and optimization of process parameters.
3. *Analysis of a generalized class-e amplifier*: this comparison focuses on simulation of electronic circuits and requires features for table functions, eigenvalue analysis and complex experiments.
4. *Dining philosophers I*: this is a more general comparison involving simulation and different modeling techniques, e.g. Petri nets.
5. *Two state model*: this comparison primarily addresses simulation tasks which require very high accuracy. It checks integration and state event handling with high accuracy.
6. *Emergency department - follow-up treatment*: this comparison addresses discrete simulation languages and tests features for modeling, concepts of availability and complex control strategies.
7. *Constrained pendulum*: this comparison is targeted at continuous simulation languages and checks features for model comparison, state events and boundary value problems.
8. *Parallel comparison*: this comparison deals with the benefits of distributed and parallel computation for simulation tasks. Three test examples have been chosen to investigate the types of parallelization techniques best suited to particular types of simulation tasks.
9. *Canal-and-lock system*: this comparison is targeted at discrete simulators and checks features for modeling complex logic, which has to be verified by deterministic data sets. Variance reduction capabilities are also checked.
10. *Fuzzy control of a two tank system*: this comparison asks for either the availability of modules for fuzzy control or for the detailed specification of how such modules can be implemented efficiently.
11. *Dining philosophers II*: this comparison reviews discrete simulators with respect to simultaneous, i.e. concurrent, access to resources.

12. *SCARA robot*: this comparison is targeted at continuous simulation languages and deals with the handling of implicit systems.
13. *Collision processes in rows of spheres*: this comparison deals with a model of the mechanics of spheres. The features to be compared represent a large number of events, numerical accuracy, the iteration of a boundary value and stochastic parameter variations. Piecewise, constant velocities permit both a continuous and a discrete treatment.

The solutions to four of the above comparisons, implemented in DSOL, are presented in this section. Two of the four are continuous models (1 and 3), one is a discrete event model (6), and one test is a continuous-discrete event hybrid model (2). The reason for selecting these particular comparisons is that besides representing well known basic problems, they illustrate some of the advantages of the N_n - N_m - N_o paradigm underlying the suite. At the moment of writing this thesis, the following tests have been implemented in DSOL and have been submitted for publication in SNE(Jacobs, 2004): 1, 2, 3, 4, 6, 8, 9 and 10. Note, where others contributed to the test they are named.

The solutions to four of these comparisons, implemented in DSOL, are presented in this section.

6.1.1 Comparison 1: lithium-cluster dynamics

The first model to be compared is taken from solid state physics. The special features to be compared are rate equations, stiff systems, parameter sweep and steady-state calculation.

The model describes the formation and decay of defect F-centers aggregates, i.e. electron centers, in alkali halides. The defects are produced by electron bombardment near the surface of the crystal and they either form aggregates or evaporate if they reach the surface.

The variable $f(t)$ denotes the concentration of F-centers, $m(t)$ and $r(t)$ respectively denote the concentration of aggregates consisting of two M-centers or three R-centers. The system can be easily extended to take into account the formation of larger aggregates. The variable $p(t)$ is the production term of F-centers due to electron bombardment, i.e. irradiation:

The features to be compared in the first model are rate equations, stiff systems, parameter sweep and steady-state calculation;...

$$\frac{dr}{dt} = -d_r r + k_r m f \quad (6.1)$$

$$\frac{dm}{dt} = d_r r - d_m m + k_f f^2 - k_r m f \quad (6.2)$$

$$\frac{df}{dt} = d_r r + 2d_m m - k_r m f - 2k_f f^2 - l_f f + p \quad (6.3)$$

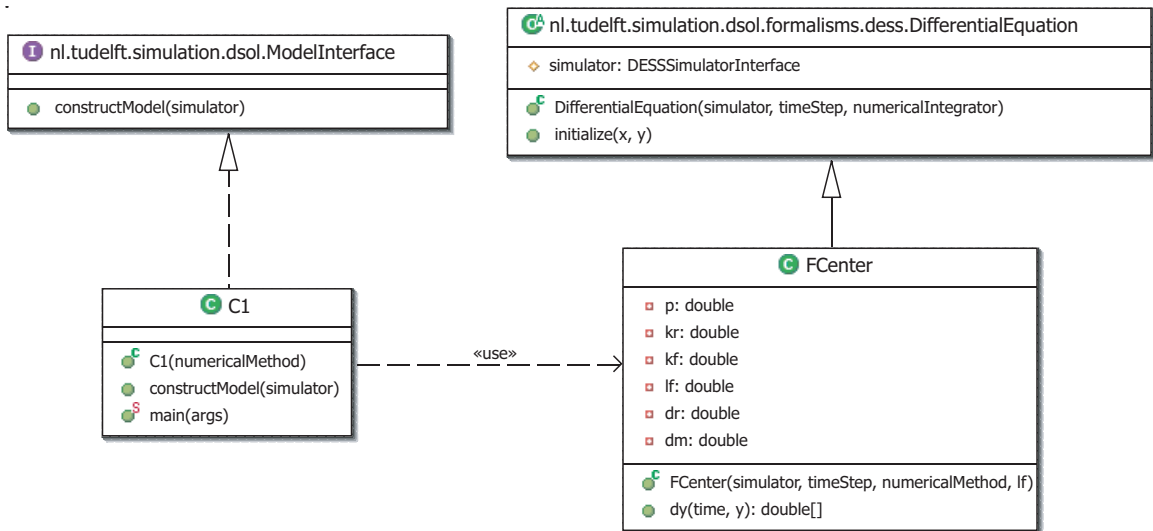


Fig. 6.1: Class diagram of F-Center

The parameter l_f measures the loss of F-centers at the surface; k_r and k_f are rate constants describing the formation of an M-center out of two F-centers, or the formation of an R-center out of an M-center and an F-center. The decay of an R-center into an M-center and an F-center is described by the rate constant d_r and the decay of an M-center into two F-centers by the rate constant d_m . Investigations are started after constant electron bombardment $p(t) = pc = 10^4$ of approximately 10 s; the production term has to be set to zero $p(t) = 0$, the initial values are: $f(0) = 9.975$, $m(0) = 1.674$, and $r(0) = 84.99$. The parameter values are: $kr = 1$, $kf = 0.1$, $lf = 1000$, $dr = 0.1$ and $dm = 1$. The following tasks should be performed:

- simulate the stiff system over $t=[0,10]$ with an indication of computing time depending on different integration algorithms
- varyate lf from 10^2 to 10^4 and a plot of all $f(t;lf)$, logarithmic steps preferred
- calculate the steady states during constant bombardment, i.e. $p(t) = pc = 10^4$, and without bombardment, i.e. $p(t) = 0$

Specification in DSOL

...this requires a continuous modeling formalism.

The F-Centers present a clear example of a differential equation and thus require a continuous modeling formalism. The class diagram of the solution is presented in figure 6.1: classes **C1** and **FCenter** are specifically designed for this comparison.

The specification of the `dy` method in the `F-center` class illustrates how differential equations are specified in DSOL:

```
1 package nl.tudelft.simulation.sne.c1;
2
3 public class FCenter extends DifferentialEquation
4 {
5     public double[] dy(double time, double[] y)
6     {
7         // we create an array containing the new dy value
8         double[] dy = new double[3];
9
10        // let's compute the particular factors.
11        double drr = this.dr * y[0];
12        double dmm = this.dm * y[1];
13        double krmf = this.kr * y[1] * y[2];
14        double kfff = this.kf * Math.pow(y[2], 2);
15        double lff = this.lf * y[2];
16
17        //now we compute and return value for dy
18        dy[0] = krmf - drr;
19        dy[1] = drr - dmm + kfff - krmf;
20        dy[2] = drr + 2 * dmm - krmf - 2 * kfff - lff + this.p;
21        return dy;
22    }
23 }
```

The `C1` class implements the `ModelInterface` and constructs the model. The numerical integrator, the number of replications and the simulator are specified in an xml-based experiment file. This experiment file, the library containing the model and the suite may well be hosted at different remote locations.

Task A: simulation of the stiff system over $t=[0,10]$

The first task is to simulate the stiff system over $t=[0,10]$ and to indicate computing time depending on different integration algorithms.

The measurements presented in table 6.1 were conducted on a 1Ghz Pentium III, 512MB RAM system with DSOL 1.6 on JRE 1.4.2. The measurements are based on 20 replications and, due to the stiffness of the ordinary differential equation, have a time step of 0.001.

The computing times of simulating the system are presented in table 6.1.

Tab. 6.1: Comparing numerical integrators

Numerical integrator	Computing time (milliseconds)	Efficiency fraction
Euler	10	$\equiv 1.0$
Heun	22	0.45
Runge Kutta 3	34	0.29
Runge Kutta 4	45	0.22
Adams	90	0.11
Milne	164	0.06
Gill	185	0.05
Runge Kutta-Cash Carp	192	0.05
Runge Kutta-Fehlberg	333	0.03

Task B: parameter variation of $1f$ from 10^2 to 10^4

The first criterium used to evaluate this task is whether the variation is specified in the model, at runtime or in a script. Since DSOL's experimentation is based on Ören and Zeigler (1979); Zeigler (1984) who present a $1..N$ relation between an experiment and a set of treatments, script and model based specification are both supported.

DSOL's introspection service furthermore provides support for the alteration of values at runtime through a graphical user interface. The statistical output of DSOL is presented in figure 6.2. This screen provides spreadsheet alike functionality; statistical objects can be dragged and dropped into the table. A second evaluation criterion is whether logarithmic plots are supported. The logarithmic y-axis in figure 6.2 is based on the following piece of code:

```

3  new XYChart(
4      simulator,
5      "F-center value",
6      XYChart.XLOGARITHMIC_YLOGARITHMIC);

```

Task C: calculation of steady states during constant bombardment

The final task in this comparison is to calculate steady states during constant bombardment $p(t) = pc = 1.0 \cdot 10^4$ and without bombardment $p(t) = 0$.

The current version of DSOL does not provide automatic steady state computation. The calculation of the steady state values are to be based on the graphical statistical output or on the average of the last n values received by the tally. The

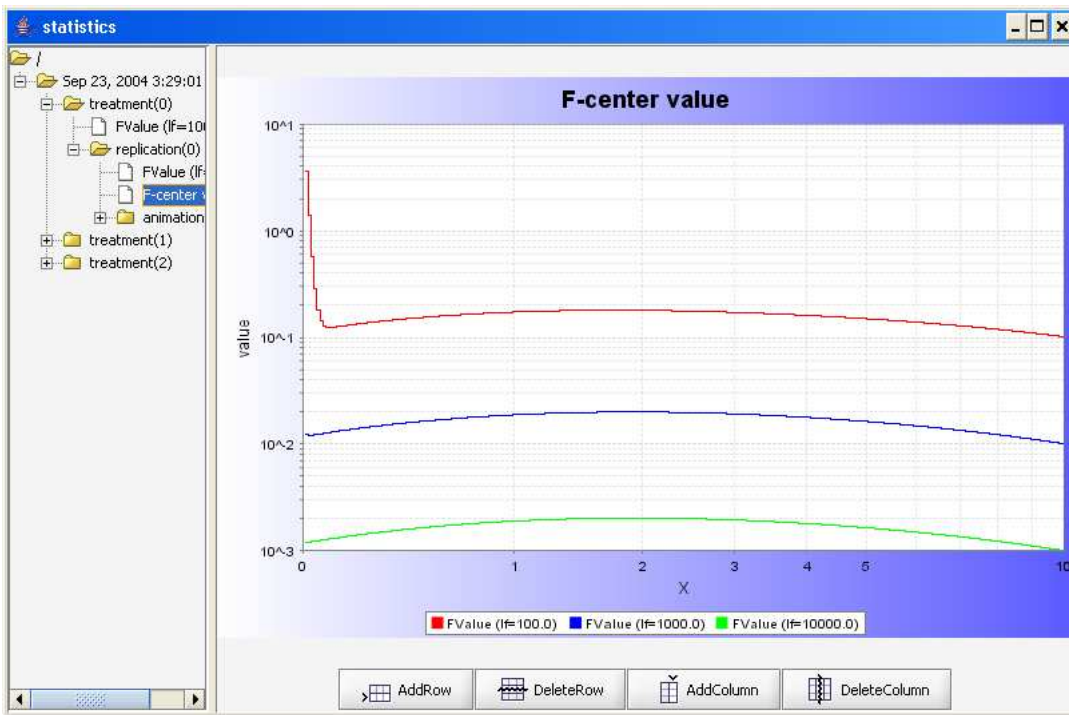


Fig. 6.2: Parameter variation presented in a logarithmic graph.

steady state values are presented in table 6.2.

Tab. 6.2: Steady state calculations in DSOL

	$p(t) = 1.0 \cdot 10^4$	$p(t) = 0$
R-centers	1000.0	0.0
M-centers	10.0	0.0
F-centers	10.0	0.0

Conclusions

The main conclusion of the specification of this comparison in DSOL is that DSOL is well suited to simulate continuous differential equation models. The continuous simulator, the abstract differential equation and the set of numerical integrators are verified. A conclusion on the $N_n-N_m-N_o$ paradigm is that the *design of contract principle* encourages future development of other numerical integrators; neither DSOL nor the model are limited to the implementations presented in table 6.1.

The main conclusion is that DSOL is well suited to simulate continuous models

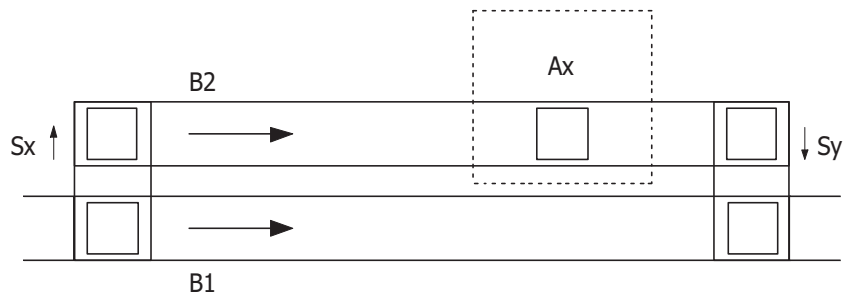


Fig. 6.3: Sub model of assembly system (Breitenecker, 2004)

6.1.2 Comparison 2: flexible assembly system

The following example of a flexible assembly system has been chosen because it checks two important features of discrete event simulation environments:

- the possibility to define and combine submodes
- the method to describe complex control strategies

The model consists of a number of almost identical submodels of the following structure (see figure 6.3). Two parallel conveyor belts, B1 and B2, are linked together at both ends. An assembly station Ax is placed at B2. Pallets come in on belt B1. If they are to be processed in Ax, they are shifted in Sx to B2 and possibly enter a queue in front of Ax. If there is no more empty buffer space on B2 or the pallet is not to be processed in Ax it continues its way along B1. Parts that have been processed in Ax are shifted back to B1 in Sy, having priority over those coming from the left on B1.

The following example of a flexible assembly system was chosen because it checks important features of discrete event simulation environments.

The total system now consists of 8 of these subsystems, varying in length, operation and operation time (see figure 6.4). Between two subsequent subsystems there is a space of 0.4 m, pallets from the third subsystem A2 can be shifted directly to A3, and from A6 directly to A1. The shifting parts, however, cannot function as buffers, i.e. a pallet can only enter an Sx if it can leave it immediately.

The operation time of each station, the total length of B1 and the length of the buffer in front of the station are presented in table 6.3.

There are three identical stations A2 in the system, because the operation in A2 takes much longer than the other operations. Unprocessed parts are put on pallets in A1. They can either be processed in A2 first, and then in A3, A4, A5, or in A3, A4, A5 first, and then in A2. The sequence of operations among A3, A4, and A5 is arbitrary. Station A6 is a substitute for any of the stations A3, A4, A5, i.e. whenever one of these stations is down, or the buffer in front of it is free, the

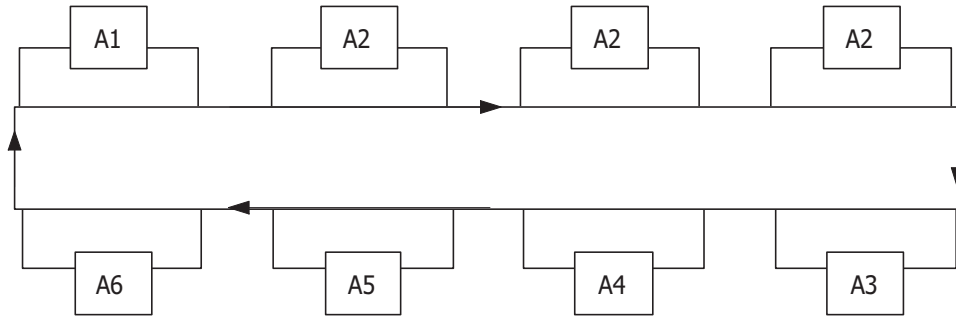


Fig. 6.4: Layout of the assembly system (Breitenecker, 2004)

Tab. 6.3: Properties of individual assembly stations

station	operation time (s)	length of B1 (m)	length of buffer (m)
A1	15	2.0	1.2
A2	60	1.6	0.8
A3	20	1.6	0.8
A4	20	1.6	0.8
A5	20	1.6	0.8
A6	30	2.0	1.2

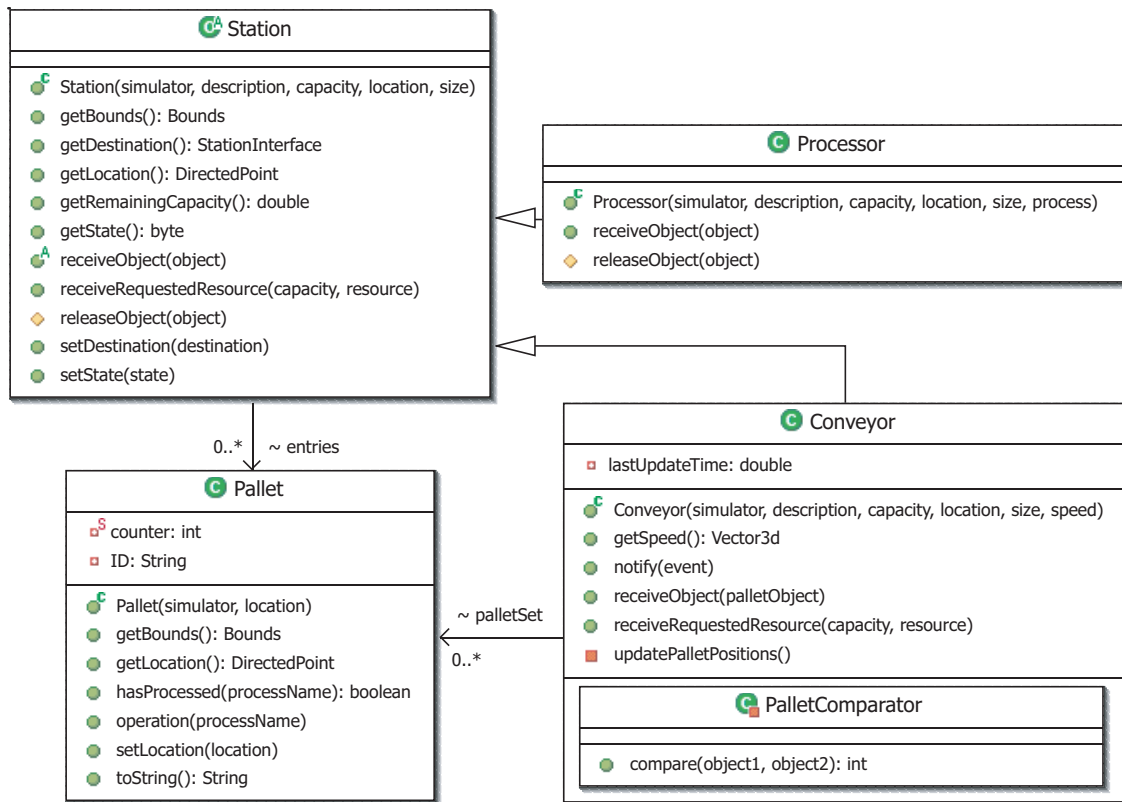


Fig. 6.5: Class diagram of DSOL specification

corresponding operation can be executed in A6. Finished parts are unloaded in A1, unfinished parts enter another circle.

All the conveyors are running at a speed of 18 m/min., any shifting takes 2 sec., and pallet length is 0.36 m. Assuming that no station ever has a breakdown, the optimum number of pallets in the system can be found. The total throughput time and the average throughput times of the parts have to be evaluated, when 20, 40 and 60 pallets are circulating in the system.

Specification in DSOL

The specification of this comparison largely reflects the work of Samson et al. (2004). The first task and corresponding evaluation criterium deals with the conceptualization and the specification of the model. In other words: How are the modular sub systems of the problem description reflected in the specification of the DSOL model?


```

130  /**
131   * Conveyors are subscribed to the TIME_CHANGED_EVENT
132   *
133   * @param event the incoming event
134   */
135  public void notify(EventInterface event)
136  {
137      if (event.getType().equals(SimulatorInterface.TIME_CHANGED_EVENT))
138      {
139          if (this.getState() != Station.SWITCHED_OFF_STATE)
140          {
141              this.updatePallets();
142          }
143      }
144  }

```

Fig. 6.6: The Conveyor.notify() method

A class diagram of the DSOL specification is presented in figure 6.5. In this diagram the `Pallet` class specifies the pallets as they are moved through the assembly system. The infrastructure in the system is modeled using the `Station` class. Its public `receiveObject` operation receives pallets and schedules their release (scheduling implies the specification in the discrete event formalism). The `Station` class extends DSOL's `Resource` class; this ensures an ability to request and seize the capacity of the station. The `Station` class is extended by the `Conveyor` class and the `Processor` class. The `Conveyor` class moves pallets through the system and has the following characteristics:

- although the entire system is specified in the discrete event formalism (DEVS), the conveyor is specified in the continuous DESS formalism. This is presented in figure 6.6: a conveyor is subscribed to time changed events and will update its pallets based on the elapsed $\delta(time)$.
- it is specified as an infrastructure component which actually introspects objects for potential, three dimensional intersection; if so these objects, i.e. pallets, are moved
- it uses an underlying binary tree with corresponding comparator, i.e. the `PalletComparator`, to sort and store pallets

Although the entire system is specified in the discrete event formalism (DEVS), the conveyor is specified in the continuous DESS formalism.

Animation is used to present the behavior of the system. A class diagram presenting the loosely coupled structure between a model component, i.e. the `Pallet`

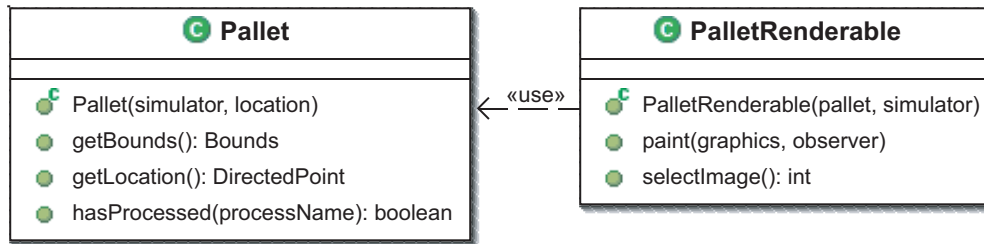


Fig. 6.7: Class diagram of Pallet animation

class, and its two-dimensional visualization component is presented in figure 6.7. The `PalletRenderable` uses the pallet to visualize a state based pallet. In this particular example the animation thus depends on the size of the pallet, on its location and on the stations that are completed.

A number of pallets are presented in figure 6.8. Since some of the side branch conveyors presented in this figure have reached their maximum of four pallets, some pallets follow the main conveyor. After a pallet is processed in the side branch, the number of the particular station is presented underneath the pallet image. Junctions, mergers and processors are colored green whenever their state is processing.

A final evaluation criterium of this particular comparison deals with the specification of the statistics in the model. The sample-mean, minimal and maximum service times are plotted as a function of the number of pallets in figure 6.9. These values correspond to the values presented by other solutions (Breitenecker, 2004). The logarithmic plots in this chart are fed by `ExitProcess` which is specified as follows:

```

2     double serviceTime = this.simulator.getSimulatorTime()
3         - ((Double) this.pallets.get(pallet)).doubleValue();
4     super.fireEvent(PALLET_SERVICETIME_EVENT, serviceTime, this.simulator
5         .getSimulatorTime());
  
```

Conclusions

In this particular comparison we have shown the strength of multi-formalism modeling. While the behavior of the `Processor` is specified in a discrete event formalism, i.e. the DEVS formalism, the behavior of the conveyor is specified in the continuous DESS formalism. We have further shown that object orientation

A final evaluation criterium of this particular comparison deals with the specification of the statistics in the model (see figure 6.9)

In this particular comparison we show the strength of multi-formalism modeling.

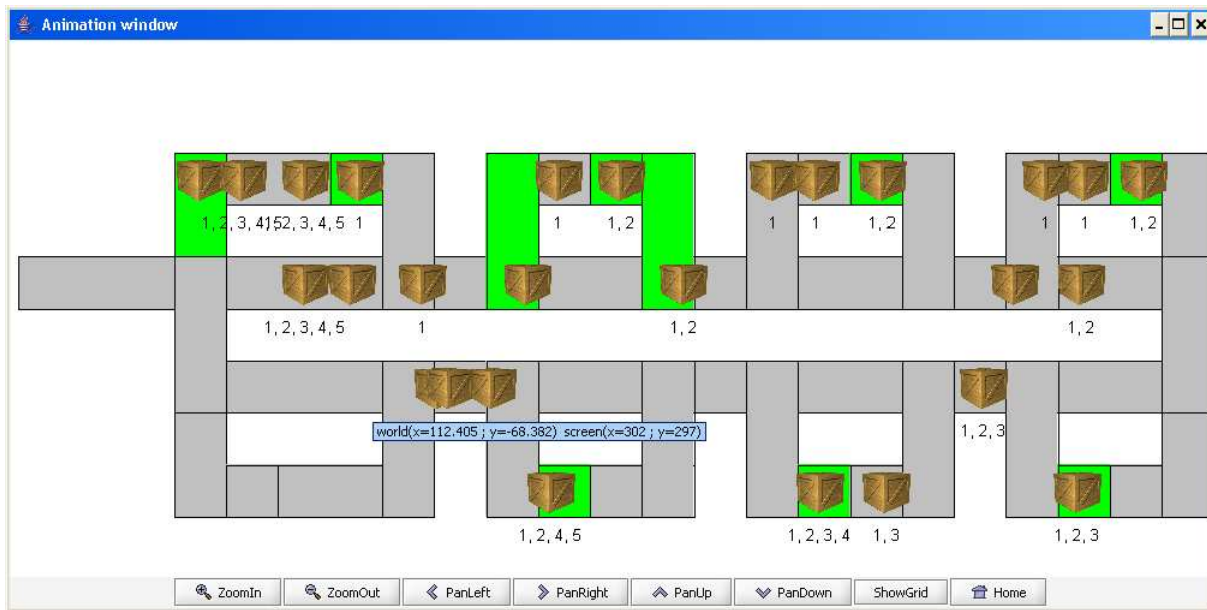


Fig. 6.8: Animation of the assembly system

enables the design of highly modular simulation models and that DSOL is well suited for the simulation and animation of this type of complex, infrastructure models.

6.1.3 Comparison 3: generalized class-E amplifier

This example is taken from the electrical engineering world. The basic class-E power amplifier was introduced by Sokal and Sokal (1975). It is a switching-mode amplifier that operates with zero voltage and zero slope across the switch at switch turn-off. The actual numerical example is taken from Mandojana et al. (1990). The component values presented in figure 6.11 are: $V_{DC} = 5$ volt, $L_1 = 79.9 \cdot 10^{-6}$ henry, $C_2 = 17.9 \cdot 10^{-9}$ farad, $L_3 = 232.0 \cdot 10^{-6}$ henry, $C_4 = 9.66 \cdot 10^{-6}$ farad and $R_L = 52.4$ ohm.

The time dependent resistor $R(t)$ models the active device acting as a switch with an ON resistance of 0.05 ohm and an OFF resistance of $5.0 \cdot 10^{-6}$ ohm. An extreme ON resistance of value zero ohm will result in a pathological system, e.g. a system behaving neurotic because it is suddenly short circuited. Furthermore the DC voltage source will be short circuited through the ideal coil L_1 . $R(t)$ is given in the graph of figure 6.11 as a function of time:

The duty ratio is 50%. The period is 10^{-6} seconds, i.e. the frequency is 100 kHz. The rise/fall time is $TRF = 10^{-15}$ seconds.

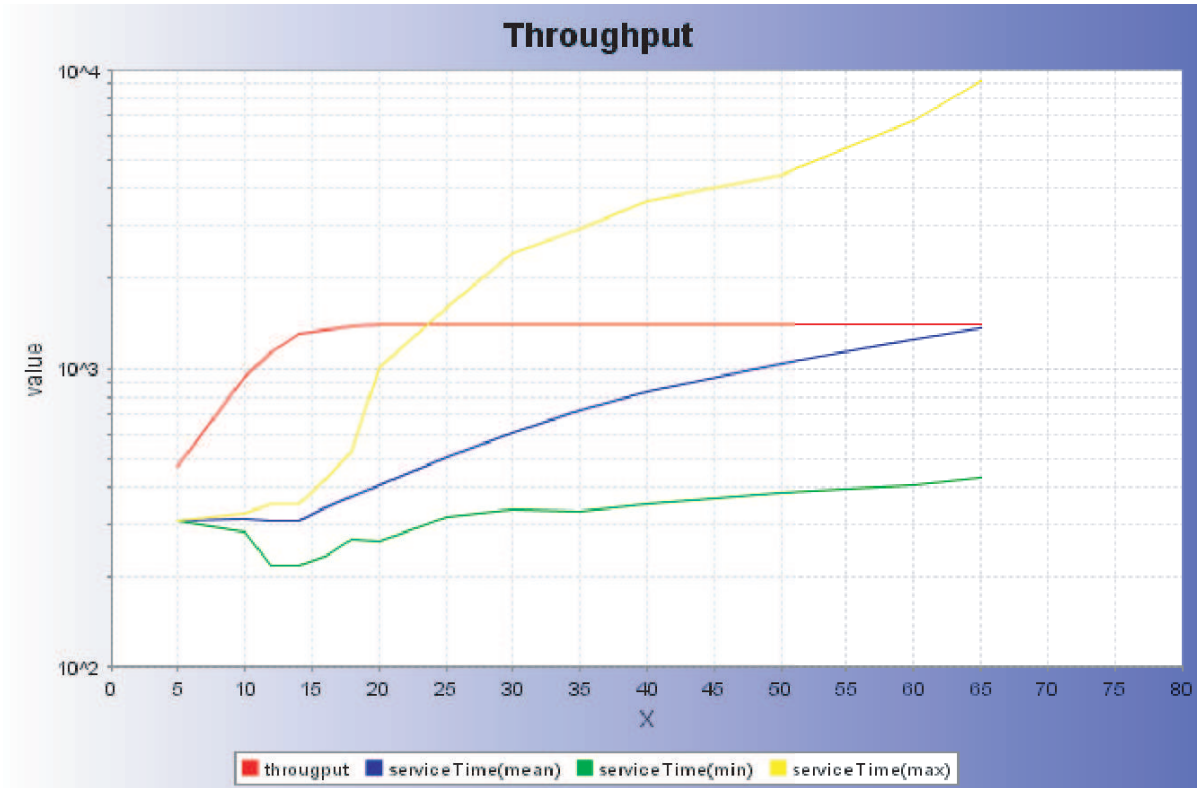


Fig. 6.9: Service times of pallets in the assembly system

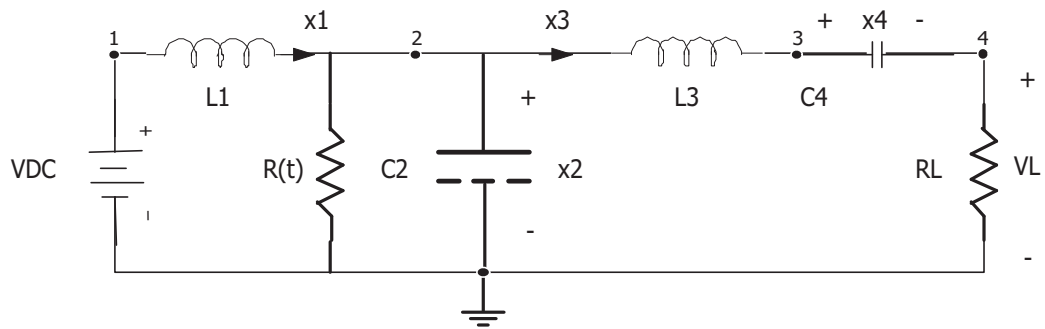


Fig. 6.10: Class-E amplifier (Breitenecker, 2004)

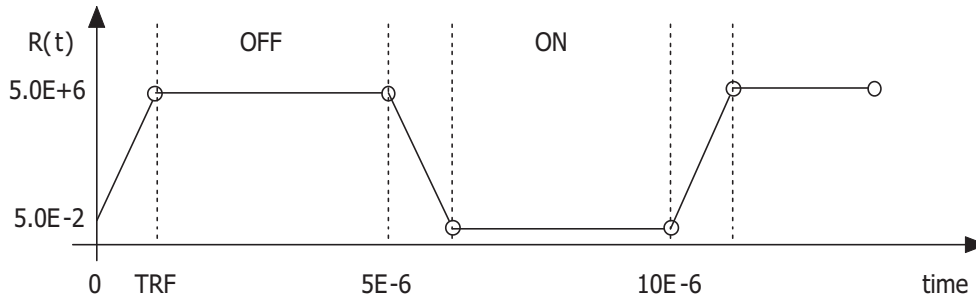


Fig. 6.11: Resistance function of the amplifier (Breitenecker, 2004)

The equations describing the circuit may be the state-equations where inductor currents and capacitor voltages are chosen as system variables. Using the Kirchoff voltage and current laws we get the following differential equations:

$$L_1 \frac{dx_1}{dt} = -x_2 + VD_C \quad (6.4)$$

$$C_2 \frac{dx_2}{dt} = +x_1 - \frac{x_2}{R(t)} - x_3 \quad (6.5)$$

$$L_3 \frac{dx_3}{dt} = +x_2 - R_L * x_3 - x_4 \quad (6.6)$$

$$C_4 \frac{dx_4}{dt} = +x_3 \quad (6.7)$$

where the variables are as follows: $x_1 = I \cdot L_1$ (the current of L_1), $x_2 = VC_2$ (the voltage of C_2), $x_3 = I \cdot L_3$ (the current of L_3) and $x_4 = VC_4$ (the voltage of C_4). Note that normally the setup of state equations demands a topological analysis of the circuit excluding some inductor currents and capacitor voltages as candidates for system variables, e.g if there is a loop of N capacitors then only N-1 of these may be given an arbitrary initial charge.

Task A: Calculation of the eigenvalues of the system

The first task is to calculate the eigenvalues of the system in the ON-period: $R(t)=0.05$ ohm and in the OFF-period: $R(t)=5.0 \cdot 10^6$ ohm. The underlying philosophy of DSOL is to provide a set of core simulation services (Jacobs et al., 2002; Jacobs and Verbraeck, 2004b). Where possible DSOL uses open source, generally

The first task of this third comparison is to calculate the eigenvalues of an amplifier,...

$$\begin{bmatrix} -3941.591492 + 833295.62856i \\ -3941.591492 - 833295.62856i \\ -108995.029583 + 661370.168244i \\ -108995.029583 - 661370.168244i \end{bmatrix}$$

Fig. 6.12: Eigenvalues of the system with R(t)=OFF=5.0 · 10⁶

$$\begin{bmatrix} -1.117318 \cdot 10^9 \\ -625.786362 \\ -112931.033798 + 658368.705292i \\ -112931.033798 - 658368.705292i \end{bmatrix}$$

Fig. 6.13: Eigenvalues of the system with R(t)=ON=5.0 · 10⁻²

available external services (requirement 4.2 on page 59). Examples include geographical information services, the Java 3D animation libraries, and in the case of eigenvalue computation, CERN's colt package.

...for which we use the COLT library developed by the European Organization for Nuclear Research (CERN).

Colt is an open source library for high performance scientific and technical computing in Java (Hoschek, 2002). Colt was developed by the European Organization for Nuclear Research (CERN). Computing the eigenvalues of the ODE is a very straightforward task. The `getEigenValues` operation of the `Amplifier` class illustrates its specification:

```

63  public EigenvalueDecomposition getEigenValues(double time)
64  {
65      double[][] matrix = {{0, -1 / l1, 0, 0},
66                          {1 / c2, -1 / (c2 * this.r.getValue(time)), -1 / c2, 0},
67                          {1 / l3, 0, -r1 / l3, -1 / l3}, {0, 0, 1 / c4, 0}};
68      return new EigenvalueDecomposition(
69          new DenseDoubleMatrix2D(matrix));
70  }
```

One of the evaluation criteria of this task is the specification of the stepped resistance function. DSOL does not provide a pre-defined Heaviside¹ step function, i.e. a function that is often used as a switch. A `Resistor` class was specified which `getValue()` function is based on a time modulus period algorithm.

¹ $H(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$

Task B: Simulation of the system over the time interval $[0, 10^{-4}]$

The second task was to simulate the system over the time interval $[0, 10^{-4}]$ sec with the zero-solution as initial state. Time curves of the state variables, the current in the switch resistor $IR(t) = x_2/R(t)$ and the output voltage $V_L = x_3 * R_L$ are wanted.



Fig. 6.14: Results of simulating the stiff system over $t=[0, 10^{-4}]$ sec.

The graphical output of the numerical integration of the stiff system is presented in figure 6.14. The same set numerical integrators presented in comparison 1 can be used. Hybrid or analytical approaches are not supported by DSOL.

The graphical output of the numerical integration of the stiff system is presented in figure 6.14.

Task C: a parameter variation study over the time interval $[0, 9 \cdot 10^{-6}]$

The final tasks was a parameter variation study over the time interval $[0, 9 \cdot 10^{-6}]$ sec with initial solution equal to the final solution at $100 \cdot 10^{-6}$ sec from task b. The rise/fall time TRF should be varied through the values: $1.0 \cdot 10^{-15}$, $1.0 \cdot 10^{-11}$, $1.0 \cdot 10^{-9}$, $1.0 \cdot 10^{-7}$ sec. The phase plane curves of $dx_3/dt = V_L/3$ as a function of $x_3 = I \cdot L_3$, i.e the voltage difference $V_2 - V_3$ as a function of the current $I \cdot L_3$ are wanted. Time curves of the current in the switch resistor $IR(t) = x_2/R(t)$ and the output voltage $V_L = x_3 * R_L$, are wanted.

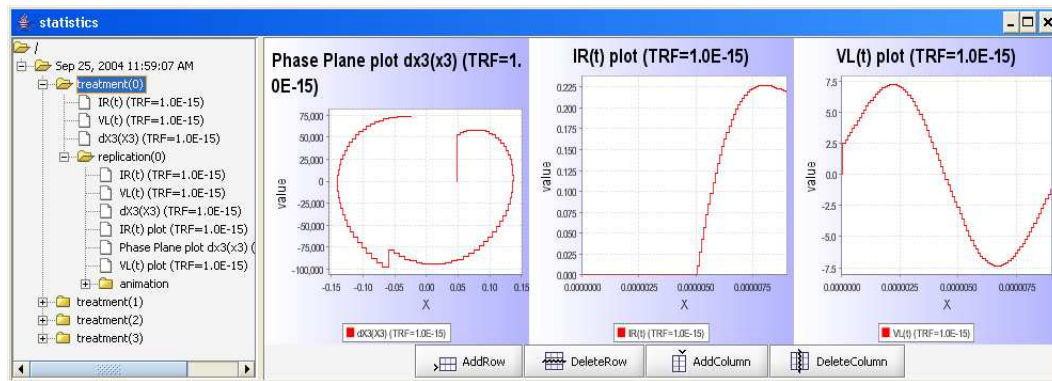


Fig. 6.15: Results of the parameter variation study over $t=[0, 9 \cdot 10^{-6}]$ sec.

The outcome of this task is presented in figure 6.15. Parameter variation is specified in the experiment definition presented in appendix 9.3. Parameter variations are thus completely automated. Object orientation makes the re-initialization of the Amplifier with the outcome of task B a straightforward task:

```

51     Amplifier amplifier = new Amplifier(
52         (DESSSimulatorInterface) simulator, 1E-15);
53
54     //We initialize with a null state and computed until 100E-6
55     amplifier.initialize(0.0, new double[]{0, 0, 0, 0});
56     amplifier.initialize(0.0,amplifier.y(100E-6));

```

Conclusions

Besides illustrating the correctness of the numerical integrators and the ability to simulate stiff systems, the first task of this comparison illustrates a unique characteristic of DSOL. The $N_n-N_m-N_o$ paradigm focuses on using external, well verified and validated services wherever possible; the computation of eigenvalues was delegated to CERN's colt package.

6.1.4 Comparison 6: emergency department

Casualties of accidents are admitted to an emergency department to have their wounds dressed. Broken limbs are put in plaster. After a few days a follow-up examination must be performed to monitor the healing process. If necessary, ad-

ditional treatment will be administered. Follow-up treatment in the emergency department of a hospital is the discrete process investigated in this comparison. The emergency department comprises the following facilities for follow-up treatment:

The process interaction formalism is chosen for the specification of this emergency department case because...

- registration where casualties are assigned to casualty wards 1 or 2; here the necessity for further treatment is established.
- waiting area, i.e. people waiting to enter casualty wards 1 and 2.
- two casualty wards CW_1 and CW_2 ; with two doctors each but with CW_2 staffed only by less experienced doctors for attention to simple cases.
- X-ray room with two X-ray units, where two patients can be X-rayed at the same time.
- a room where plaster casts are applied or removed.

Patients start arriving at 7.30 a.m. and queue for registration; doctors start work at 8.00 a.m. They attend to four types of patients.

1. Patients requiring X-rays. These patients are first examined in the casualty ward, then sent to the X-ray room, and before they leave their X-ray photographs are examined once again on the casualty ward.
2. Patients requiring removal of plaster casts. These patients enter a casualty ward, are sent to the plastering room, then leave the department.
3. Patients with plaster casts requiring an X-ray and renewal. These patients enter the casualty ward, are sent to the X-ray room and given new plaster casts. After the new plasters are checked by X-ray patients are readmitted to the casualty ward to hear the results of their X-ray from a doctor. They then leave the department.
4. Patients needing wound dressings to be changed. These patients are admitted to a casualty ward, the dressing is changed and then leave the department.

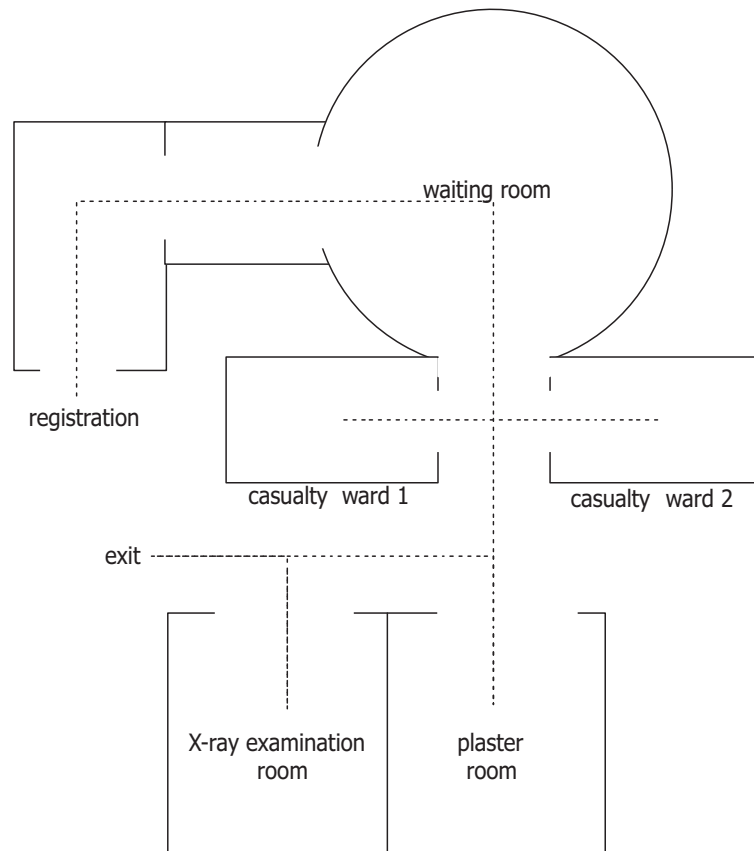


Fig. 6.16: Layout of the emergency department (Breitenecker, 2004)

Tab. 6.4: Triangular distributions of process times

Process	minimum	average	maximum
Registration	0.2	0.5	1.0
CW_1	1.5	3.2	5.0
CW_2	2.8	4.1	6.3
X-ray	2.0	2.8	4.1
Plaster	3.0	3.8	4.7

The statistical parameters are as follows:

- the time between arrivals of patients is distributed exponentially with parameter 0.3 minutes.
- the percent distribution of patients over the four groups described above is as follows: 1: 35%, 2: 20%, 3: 5%, 4: 40%.
- 60% of patients waiting for admission to a casualty ward are admitted to ward CW_1 , 40% to CW_2 . The parameters of the single treatment points show a triangular distribution presented in table 6.4.
- patients wait in queues before every treatment point.

Task A: determine average overall treatment time

The first task is to determine the average overall treatment time for 250 patients and to classify these patients by types 1 to 4. The evaluation criteria are the specification of the control and the classification of patient. The process interaction formalism is chosen for the specification in DSOL because the conceptual introduction of the case presents independent autonomous processes of patients (see figure 6.17). The output presented in table 6.5 reflects the following specification of the `process` method:

...the conceptual introduction of the case presents independent autonomous patient processes.

```

36 public class Patient extends Process implements ResourceRequestorInterface
37 {
49     private EmergencyDepartment emergencyDepartment = null;
52     private int type = -1;
61     public Patient(final EmergencyDepartment emergencyDepartment,
62                 final DEVSSimulatorInterface simulator, final int type)
63     {
64         super(simulator);

```

```

65     this.emergencyDepartment = emergencyDepartment;
66     this.type = type;
67 }
68
72 public void process() throws SimRuntimeException, RemoteException
73 {
74     double startTime = super.simulator.getSimulatorTime();
75     // we request registration
76     Department registration = this.emergencyDepartment.getRegistration();
77     registration.requestCapacity(1.0, this);
78     this.suspend();
79
80     // now we are being registered
81     this.hold(registration.getServiceTime().draw());
82
83     // let's release registration
84     registration.releaseCapacity(1.0);
85
86     // Let's claim a doctor
87     CasualtyWard casualtyWard = this.emergencyDepartment.getCasualtyWard();
88     casualtyWard.getDoctor().requestCapacity(1.0, this);
89     this.suspend();
90
91     // now we are being helped
92     this.hold(casualtyWard.getServiceTime().draw());
93     casualtyWard.getDoctor().releaseCapacity(1.0);
94
95     // Let's see what to do
96     switch (this.type)
97     {
98         case 1 :
99             this.xRay();
100            break;
101            case 2 :
102                this.plaster();
103                break;
104            case 3 :
105                this.xRay();
106                this.plaster();
107                this.xRay();

```

```

108     // we claim the same ward again
109     casualtyWard.getDoctor().requestCapacity(1.0, this);
110     this.suspend();
111
112     // now we are being helped
113     this.hold(casualtyWard.getServiceTime().draw());
114     casualtyWard.getDoctor().releaseCapacity(1.0);
115     break;
116 }
117 System.out.println(this.simulator.getSimulatorTime() - startTime);
118 }
119
120 /**
121  * processes type 1 patients
122  *
123  * @throws RemoteException on remote failure
124  * @throws SimRuntimeException on simulation failure
125  */
126 protected void xRay() throws RemoteException, SimRuntimeException
127 {
128     // Let's claim the xRay
129     Department xRay = this.emergencyDepartment.getXRayRoom();
130     // xRay.requestCapacity(1.0, this,Resource.MAX_REQUEST_PRIORITY);
131     this.suspend();
132
133     // now we are being xRayed
134     this.hold(xRay.getServiceTime().draw());
135     xRay.releaseCapacity(1.0);
136 }
137
138 protected void plaster() throws RemoteException, SimRuntimeException
139 {
140     //plaster the patient
141 }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }

```

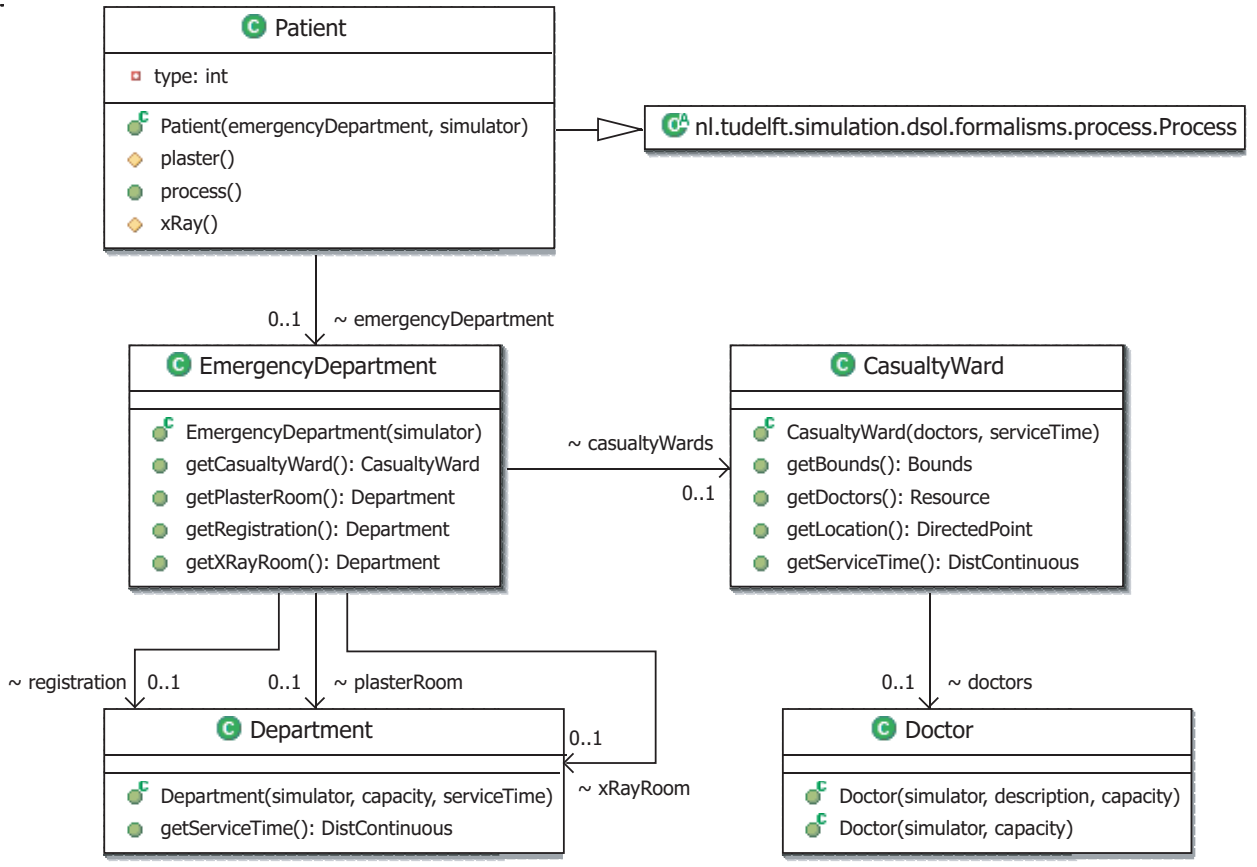


Fig. 6.17: Class diagram of the emergency department case

Tab. 6.5: Task A: overall treatment times

Type	Average treatment time (minutes)
1	227
2	164
3	223
4	159
overall	201

Task B: assume that an experienced doctor from CW_1 replaces one of the inexperienced doctors in CW_2

Assume that an experienced doctor from CW_1 replaces one of the inexperienced doctors in CW_2 as soon as the queue for CW_2 exceeds 20 patients. The CW_1 doctor moves to CW_2 to continue working as well as he can. Note that the working time of the doctor from CW_2 now working in CW_1 is increased by 20% due to the more complex cases he or she has to deal with. As soon as the queue for CW_2 is down to five people the inexperienced doctor still working in CW_1 is returned to CW_2 . The alteration of the model with respect to this assumption is straightforward in DSOL. DSOL's `Resource` class extends the `EventProducer` class and fires events on length changes of the queue. The subscription by the experienced doctors on this particular event type accomplishes the task.

Specification of the assumption that an experienced doctor from CW_1 replaces one of the inexperienced doctors in CW_2 ...

...is straightforward in DSOL, because the use of the asynchronous event package.

```
8 public class ExperiencedDoctor extends Doctor
9 implements EventListenerInterface
10 {
15     public ExperiencedDoctor(final DEVSSimulatorInterface simulator,
16         final double capacity, final Doctor juniorColleagues)
17     {
18         super(simulator, capacity);
19         this.juniorColleagues = juniorColleagues;
20         this.juniorColleagues.addListener(this,
21             Resource.RESOURCE_REQUESTED_QUEUE_LENGTH);
22     }
24     public void notify(final EventInterface event)
25     {
26         if (event.getType().equals(Resource.RESOURCE_REQUESTED_QUEUE_LENGTH))
27         {
28             long queueLength = ((Number) event.getContent()).longValue();
29             if (this.helping && queueLength <= 5)
30             {
31                 // undo the prior swap
32             } else if (queueLength > 20 && !this.helping)
33             {
34                 // start with the swap.
35                 this.setCapacity(1.0);
36             }
37         }
38     }
39 }
```

The results show little impact on the overall treatment time: the doctors replacement occurred one time and the two doctors changed back to their original casualty wards before the simulation was stopped. This shows clearly that changing the logic in this way does not improve the treatment time.

Task C: to minimize the standard deviation of overall treatment time by introducing a priority ranking.

Try to minimize the standard deviation of overall treatment time by introducing a priority ranking. Patients entering one of the treatment points for the second time, i.e. type 1 and type 3 patients, rank higher in priority than all other patients. The specification in DSOL of this task is straightforward. Whenever an amount of capacity is requested from a resource, a priority can be assigned to this request; no dedicated engineering was required for this task. This functionality is presented in the partial class diagram of DSOL's Resource class in figure 6.18.

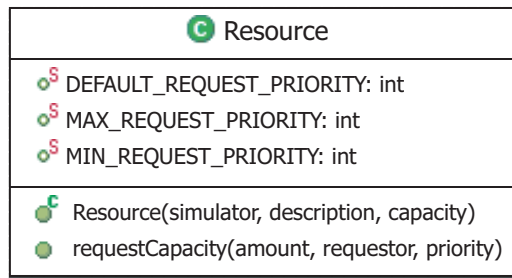


Fig. 6.18: Partial class diagram of DSOL's Resource class

Conclusions

The specification of this comparison clearly verifies the specification of the process interaction formalisms in DSOL.

The specification of the emergency department comparison clearly verifies the specification of the process interaction formalisms in DSOL. The resource library furthermore facilitates a prioritized resource allocation scheme. By using an object-oriented programming language, the identification of individual instances and types of patients is inherently supported.

6.1.5 General conclusions on the SNE comparisons

An overview of the implementation of the SNE comparisons is presented in table 6.6. The content of this table represents the solutions published on the SNE web-

Tab. 6.6: Overview of the implementation of SNE comparisons

Simulation environment	1	2	3	4	5	6	7	8	9	10	11	12	total
ACSL	•		•		•		•		•		•	•	7
AnyLogic	•	•		•		•	•		•	•	•	•	9
Arena		•		•		•				•			4
Automod		•											1
CSIM		•								•			2
Desire	•		•				•						3
Desmo		•		•									2
DOSIMIS		•											1
Dymola					•		•		•		•	•	5
Dynast	•		•										2
Enterprise Dynamics		•											1
EXTEND	•	•											2
GPSS		•		•		•		•		•			5
IDAS	•		•										2
Maple	•		•		•		•		•			•	6
Mathematica	•		•				•						3
MATLAB	•		•		•		•		•	•	•	•	8
MicroSaint		•				•		•		•			4
MOSYS	•				•		•						3
MATRIX	•						•		•				3
NAP2	•		•										2
PowerSim	•		•										2
Prosign	•		•										2
SDX	•						•						2
SIL	•		•										2
Simnon	•		•				•						3
SIMUL_R	•	•	•	•	•	•	•	•					8
Simple++		•				•				•			3
SLAM		•				•							2
SLX		•				•		•		•		•	5
STEM	•		•		•		•						4
Taylor		•		•		•				•			4
WITNESS		•											1
DSOL	•	•	•	•	•	•		•	•	•			9
total (34)	20	17	15	7	8	10	14	5	6	10	5	7	

page (Breitenecker, 2004). A number of conclusions can be drawn. One, DSOL is positioned at the top end with respect to the number of solutions provided for the SNE comparisons. Two, DSOL supports multi-formalism simulation; both continuous and discrete simulation is supported. Three, the resemblance between patterns of bullets between different environments reflects the fact that these environments are either targeted at discrete or continuous simulation.

A more general conclusion is that in solving the SNE comparisons we have used external services, e.g. CERN's colt service; we have thus provided the first scientific evidence for the value of a service oriented simulation suite.

6.2 Testing and analyzing the DSOL suite

The testing and analyzing of the DSOL suite forms the topic of this section.

The testing and analyzing of the DSOL suite forms the topic of this section. Numerous tools and techniques are presented in this section which are all based on the accepted claim that test-driven development, or test-first programming, improves software quality and programmer confidence (Kaufmann and Janzen, 2003). The use of these tools results from requirement 4.11 on page 61.

6.2.1 Code formatting and style checking

Checkstyle automates the process of checking the style of Java source code.

One of the tools used throughout the development process of DSOL is *Checkstyle*; this is a development tool which helps programmers to write Java code that adheres to a strictly defined coding standard. It thus automates the process of checking the style of Java source code by enforcing a rigid coding standard (Burn, 2003). To illustrate the value of code formatting, we consider a simple example in which two methods are specified, both of which divide two numbers, *a* and *b*.

```
3 public static double DO(double a, double B)
4 {if (B!=0)return a/B;
5   throw new RuntimeException("ILLEGAL");}
6
7
8 /**
9  * divides a by b
10 *
11 * @param a a represents the first number
12 * @param b a represents the number by which a will be divided
13 * @return the fraction a/b
14 */
15 public static double divide(final double a, final double b)
16 {
```

```

17     // Let's prevent a division by zero
18     if (b != 0)
19     {
20         return a / b;
21     }
22     // we must throw an appropriate exception
23     throw new IllegalArgumentException("Cannot divide" + a
24         + " by b since b=0.0");
25 }

```

Although the first method named `DO` (lines 3-5) returns exactly the same value as the method `divide` (lines 8-25) when invoked with two numbers, the latter is much easier to understand. Checkstyle enforces code that adheres to a strictly defined coding standard and as a result would enforce the problems illustrated in figure 6.19 to be solved.
















	Missing a Javadoc comment.	Test.java	test	line 3
	Name 'B' must match pattern <code>^[a-z][a-zA-Z0-9]*\$</code> .	Test.java	test	line 3
	Name 'DO' must match pattern <code>^[a-z][a-zA-Z0-9]*\$</code> .	Test.java	test	line 3
	Parameter a should be final.	Test.java	test	line 3
	Parameter B should be final.	Test.java	test	line 3
	'!=' is not followed by whitespace.	Test.java	test	line 4
	'!' is not preceded with whitespace.	Test.java	test	line 4
	'/' is not followed by whitespace.	Test.java	test	line 4
	'/' is not preceded with whitespace.	Test.java	test	line 4
	'{' is not followed by whitespace.	Test.java	test	line 4
	'if' construct must use '{}'	Test.java	test	line 4
	'if' is not preceded with whitespace.	Test.java	test	line 4
	'return' is not preceded with whitespace.	Test.java	test	line 4
	';' is not followed by whitespace.	Test.java	test	line 5
	'}' is not preceded with whitespace.	Test.java	test	line 5

Fig. 6.19: An example illustrating the use of Checkstyle

A coding standard results in good quality code with respect to human readability and standardization (Joy et al., 2000). The standard enforces naming conventions, visibility conventions to ensure encapsulation and size conventions, i.e. line width, method length and file size. A report on the extent to which DSOL complies to style standards is published at <http://www.simulation.tudelft.nl/dsol/dsol/checkstyle-report.html>.

6.2.2 Unit testing

Another tool used throughout the development of DSOL is *Unit* testing. This implies testing individual software units or groups of related units (Link and Frolich, 2003). The rationale behind Java Unit testing is that without automated testing, it is time consuming and difficult to ensure that changes will not break existing code. For Java programmers, JUnit makes such testing easy (Morris, 2003).

Unit testing implies testing individual software units or groups of related units.

A unit test ensures that individual software components produce expected results, thus fulfilling their contract. All DSOL services are accompanied with corresponding JUnit testing classes. An example of a JUnit test is presented in appendix 9.3 on page 186. The use of `Assert` statements which are to be evaluated by the JUnit suite is presented here.

A report on the results of DSOL's JUNIT tests is published at <http://www.simulation.tudelft.nl/dsol/event/junit-report.html>.

6.2.3 Profiling

A *profiler* is a computer program that can track the performance of another program by checking informations collected while the code of the second program is executed. A profiler can identify the time used by or frequency of use of various operations of the second program. The rationality of using a *profiler* is that a suitable performance analysis tool supports the tracing of the following types of problems:

A profiler mainly supports the tracing of semantic incorrectness and inefficient methods.

- semantic incorrectness: methods have a semantic meaning which can be checked through profiling. For example, the number of times the `generate` method is invoked on a customer generator indicates the semantic correctness of a `Generator` class.
- inefficient methods: methods coded for flexibility and generality can cause significant performance problems when used extensively in larger applications. For example, developers often discover that an inordinate amount of time is spent in Java `String` methods.
- memory management: Kazi et al. (2000) explain that Java's garbage collection appears to have been optimized for applications with a relatively small memory footprint. As a result, large applications can experience unacceptably large and unpredictable delays during garbage collection.

To illustrate the value of profiling DSOL, we recall the process interaction example presented in section 5.6.2 on page 81 in which we consider boats entering a port. Profiling gives insight in both the performance loss and the correctness of

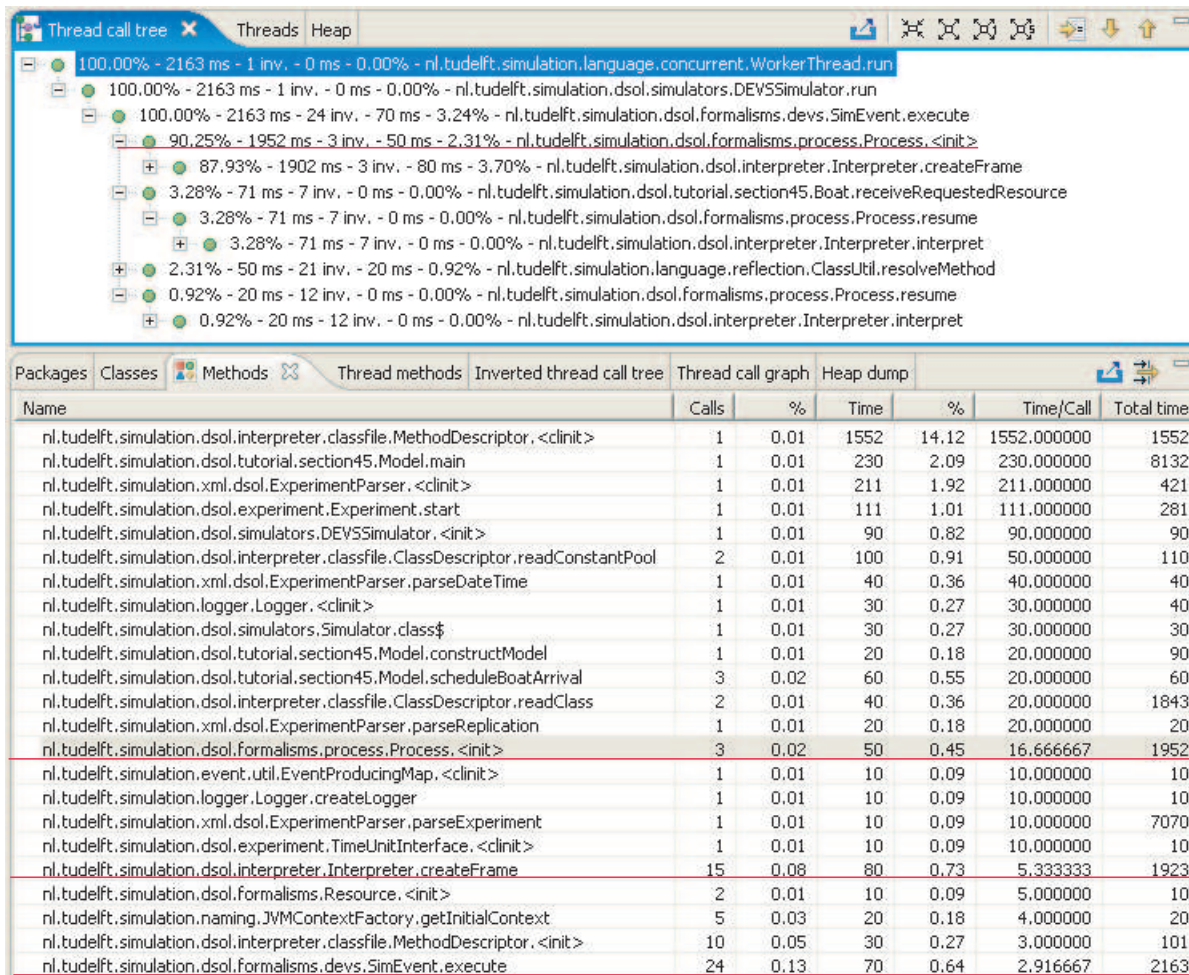


Fig. 6.20: Results of profiling a simple process interaction model

implementation. The results of profiling this particular example are presented in figure 6.20. The following facts presented in this figure are worth discussing.

- The thread call tree for the `WorkerThread` is presented in the upper part of figure 6.20. This thread is 100% dedicated to the execution of the `DEVSSimulator.run()` method. The thread call tree furthermore shows that the `Interpreter` and the `Resource` classes are used for the execution of this model.
- 90.25% of the execution time of this `WorkerThread` is spent on the `Process.<init>` method which represents the constructor of the `Process` class. This implies that the performance loss of the process interaction formalism is mostly related to the initialization of the model. This is in line with what we expected; the real Java virtual machine has to load 300 classes of the interpreter; the interpreter then has to re-parse the `Boat` class.
- The constructor of the `Process` class is invoked 3 times by the `WorkerThread`. This suggests that the process interaction formalism is indeed single threaded.
- The final line illustrated in figure 6.20 shows that 24 instances of the `SimEvent` class are executed; this proves that the process interaction formalism is embedded in the DEVS formalism (see section 4.3 on page 49).

Although the example presented in this section was chosen subjectively, we have shown the value of applying a profiler on both the DSOL suite and on particular models for the tracing of problems and the improvement of code with respect to its efficiency.

6.3 Conclusions

We presented the verification of DSOL in this chapter. We based this verification on two aspects: a verification based on SNE's comparisons and a quality analysis based on software testing and analyzing techniques.

We assumed in this chapter that the collection of comparisons presented by (Breitenecker, 2004) consists of the combined experts opinions on the completeness and correctness of a simulation environment. We specified a wide range of discrete, continuous and hybrid problems in DSOL and have verified that the output corresponds to the output presented in (Breitenecker, 2004). We thus conclude that DSOL is verified for the domain of simulation.

We showed how DSOL was linked to an external mathematical library for the computation of eigenvalues of a system in section 6.1.3; and how DSOL is linked to an external profiler in section 6.2.3. How we used an external software package

We conclude that DSOL is verified for the domain of simulation

to check the extent to which both the models created in DSOL and the core of DSOL comply to pre-defined coding standards is discussed in section 6.2.1.

Where the usefulness of a traditional simulation environments is limited to its rigid set of provided functionalities, the DSOL suite is, because of its open structure, well suited to be linked to any Java based external library. Since none of the above external services were aimed at the domain of simulation, we can draw the conclusion that it is proven that using a simulation suite provides more useful decision support than traditional simulation environments.

The quality of the suite was furthermore assessed using several verification tools to fulfill the requirement 4.11 listed on page 61 which states that software engineering verification tools should be applied to verify the quality of the DSOL source code. This correspondence and completeness check forms the basis for the validation of the DSOL suite presented in chapters 7 and 8.

This conclusion forms the basis for the real-life validations presented in chapters 7 and 8.

7. CASE: EMULATION WITH DSOL

7.1 Introduction

A case study conducted for TBA Nederland is presented in this chapter. This case study is presented as an instrument in the validation of DSOL with respect to its effectiveness for simulation; we will draw some first conclusions on the usefulness, usability and usage of DSOL at the end of this chapter. We aim to present scientific evidence that a simulation suite indeed provides more effective decision support.

TBA is a simulation consulting firm active in the domain of logistic business process (re)engineering using simulation and *emulation* (TBA Nederland, 2000). TBA's areas of expertise includes (re)design of manufacturing processes, airport capacity systems, i.e. luggage and cargo systems and container terminals.

The case presented in this chapter concerns an emulation study conducted for Dycore b.v., a concrete floor manufacturer. The value of this case for our research results from the conditions under which it was conducted: the case study was done as a competition between a team of developers from TBA and a team from Delft University of Technology. TBA's team, which consisted of two senior engineers, used their de-facto simulation environment, eM-Plant¹, while the TU Delft team, which consisted of the author of this thesis, used DSOL. The idea behind this real life, real time competition, was for TBA to be able to assess to what extent DSOL is a serious alternative to eM-Plant and whether they might wish to use it in the future for emulation projects. For our research, this case provided us with a means to make a good comparison of DSOL with a commercial simulation environment in a realistic operational setting. As we will show throughout this chapter, the conditions at Dycore provided the base for a useful case for a comparison and validation of DSOL.

We introduce the case in section 7.1.1, then we discuss the importance of *emulation* in the design of control systems in section 7.1.2. We conclude the introduction of this chapter with a requirement analysis.

A conceptual model of the case is presented in section 7.2. The emulation bottleneck, i.e. the communication between the realtime system and the simulation

The case presented in this chapter concerns an emulation study conducted for Dycore b.v., a concrete floor manufacturer.

¹ eM-Plant is an object-oriented simulation environment trademarked by Technomatix (<http://www.emplant.de>)



Fig. 7.1: Sheet piling floors (Dycore, 2004)

model, forms the topic of section 7.2.1. The specification of the model in DSOL and a comparison with the specification in em-Plant is presented in section 7.3. After presenting the experiments in section 7.3.4, we conclude this chapter with conclusions on DSOL's applicability in the domain of emulation. Note, this chapter has been read and its content has been endorsed by TBA Nederland and Dycore. The content of this chapter reflects the paper '*Emulation with DSOL*' (Jacobs et al., 2005b).

7.1.1 60m^3 of concrete floors on an automated guided vehicle

Dycore is a concrete floor manufacturer that produces annually more than $3,000,000\text{m}^3$ of concrete floors for the Dutch and global markets. Dycore employs more than 500 employees in their combined facilities. All their production plants are KOMO² certified and comply to the NEN ISO 9001³ standards.

The case presented in this chapter deals with the production of *sheet piling floors* at the manufacturing plant in Breda, the Netherlands (see figure 7.1).

The production of sheet piling floors is fully automated. All machines, sensors, conveyors, cranes and vehicles are controlled by a *programmable logic controller*, or PLC. A partial layout of the factory in Breda is presented in figure 7.2. This part is called the *bewapeningsomloop* which is best translated as the *reinforcement gallery*.

In figure 7.2, 10 rectangles represent locations where pallets containing up to 18 floors can be positioned. Horizontal movement from positions 208 to 205, 201 and 200 is effected using motors named M208, M2051, M2012, etc. Vertical movement of the pallets is achieved by automatic guided vehicles called *pups*⁴. Three of these pups move synchronously from position 207 to 204 and from position 203 to

² KOMO is a hall-mark of the SBK foundation (<http://www.komo.nl>).

³ NEN is the Dutch institute for normalization (<http://www.nen.nl>).

⁴ Pup is a direct translation of the Dutch word *hondje*, or little young dog.

The case presented in this chapter deals with the production of *sheet piling floors* at the manufacturing plant in Breda, the Netherlands.

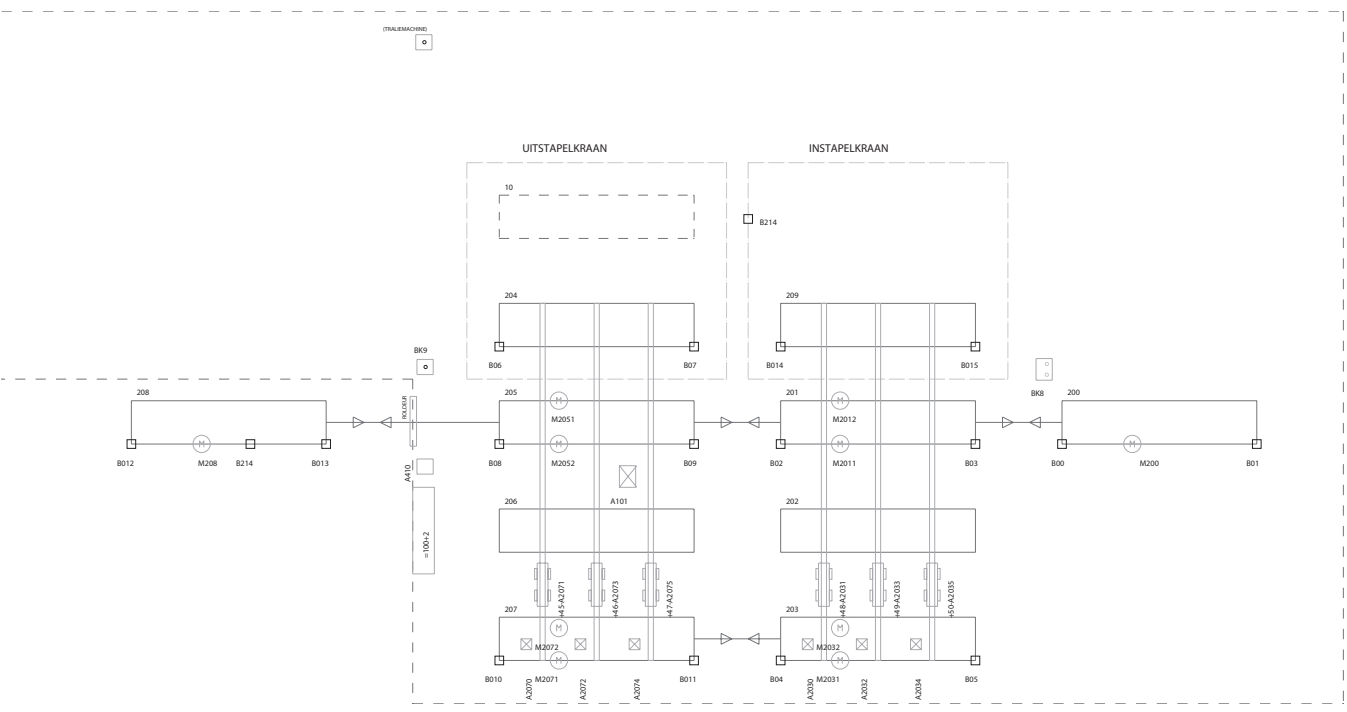


Fig. 7.2: Layout of reinforcement gallery (Dycore, 2004)

209. Besides the motors and the pups, a number of other input/output objects are illustrated. The small rectangles in the corner of each pallet location represent sensors and BK8, BK9 and A410 represent manual emergency stops. Two stacking cranes place and remove the pallets from position 204, respectively position 209. Dycore commissioned TBA to test a newly designed and developed programmable logic controller. The aim of this commissioning was to debug the engineered system in the lab - not on the factory floor, for the following reasons. One, the two shift production that is realized at this plant should not be disturbed; testing can only take place during more costly night and weekend hours. Two, operators should be trained in an environment in which the actual operation is not to be disturbed.

The aim was to debug an engineered control system in the lab - not on the factory floor.

7.1.2 The importance of emulation in the design of realtime control

Testing the behavior of a PLC, which controls one or more devices that are part of a logistic system, is usually done by connecting the PLC to stand-alone versions of each individual device, called mock-ups (Schludermann et al., 2000; Schiess, 2001). This approach to device testing is expensive and the test conditions are hard to reproduce. Above all these test are incomplete since the interaction of a device with other devices is ignored and the system as such cannot be tested as a whole (Schludermann et al., 2000; Schiess, 2001).

Emulation is a hardware-in-the-loop approach to simulation that is designed to test the behavior of a real (control) system.

Emulation is a *hardware-in-the-loop* approach that is designed to solve this incompleteness. Emulation implies that all *inputs* and *outputs* of a controller are connected to *simulated devices*. This enables better reproduction of test conditions and allows the tester to reproduce the interaction of various parts of the complete system (Schludermann et al., 2000). Whorter et al. (1997) state that using the same model for system development, system testing using emulation and staff training, can reduce costs and plant set-up times.

7.1.3 Expectations for the case

We expected DSOL to perform better than its traditional counterpart, i.e. eM-Plant, for a number of reasons which are given.

In this particular case, the emulation model is used to test the behavior of a PLC, which controls one or more devices that are part of a logistic system.

- The service oriented, open architecture underlying DSOL should make the deployment of an emulation model in a distributed, networked environment more straightforward. It should, in other words, be more straightforward to link DSOL to external components, e.g. a PLC.
- The multi threaded, scalable characteristics of the Java programming language should make DSOL more effective in the performance-defiant domain of emulation.

- The support for CAD drawings in combination with the inclusion of Java's 3D modeling should give DSOL advantage with respect to the straightforwardness of infrastructure modeling.
- DSOL clearly distinguishes a model from a simulator. This distinction supports the usage of one model for system development, system testing and training purposes.

The value of this case for the validation of DSOL was is found in the opportunity provided support these claims in a real time, real life case. A specification of the requirements for the emulation model is given below.

7.1.4 Requirement analysis

To help us understand the requirements for an emulation model of the reinforcement gallery, we will first briefly introduce the internal structure of a PLC. The major components of a PLC are its *CPU*, its *memory*, its *power supply*, its *inputs* and its *outputs*: where inputs provide a PLC with the ability to read signals from different input devices, e.g. sensors and buttons, and outputs provide a way for a PLC to control output devices, e.g. motors and cranes.

Requirement 7.1 *The first and most important requirement for the case is that the emulation model may not impose any modifications to the PLC for testing purposes. Such modification would conflict with the nature of emulation which is to test the actual system.*

Data exchange between a PLC and physical devices is based on special *industrial protocols*. This leads to a second requirement for our emulation model.

Requirement 7.2 *The emulation model should support the industrial data exchange protocol used by Dycore's PLC.*

Memory in a PLC can be distinguished into *system memory* and *user memory*. System memory is used by a PLC for its internal process control system. The user memory contains a user program in a binary form. User memory is divided into blocks having special functions. Some parts of the memory are used for storing input and output status. The real status of an input is stored either as "1" or as "0" in a specific memory bit. The combination of 16 bits is called a *word*, and each input or output has one or more corresponding bits in memory. An example of two lamps is presented in figure 7.3 to illustrate how memory is used to control specific devices. Other parts of memory are used to store variable contents for variables used in user program. For example, timer value, or counter value would be stored in this part of the memory (Matic, 2001).

The emulation model should support the industrial data exchange protocol used by Dycore's PLC.

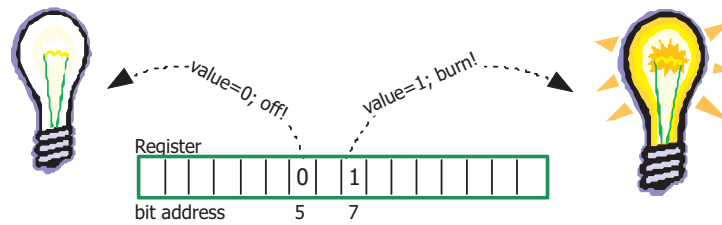


Fig. 7.3: An example of controlling two lamps.

A PLC reads its inputs and sends its outputs on a regular basis. This time interval is called the *period* of the PLC. Since this period defines the accuracy of the control system and as such of the controlled devices, a second requirement is that

The emulation model should meet the realtime period of the PLC.

Requirement 7.3 *The emulation model should meet the realtime period of the PLC.*

To ensure that an emulation model meets this real time constraint, emulation requires an underlying real time operating system (Schludermann et al., 2000).

Requirement 7.4 *In situations where an emulation model is nevertheless deployed on standard, i.e. non real time, operating systems, e.g. Linux or Microsoft Windows, the emulation environment should report backlog as it occurs.*

Where both requirements focus on the usefulness of the emulation model, TBA presented a further set of requirements with respect to the usability of the testing environment.

The emulation model should animate all devices in realtime on top of the CAD layout. All simulated devices should further be controllable at runtime through a graphical user interface.

Requirement 7.5 *The emulation model should animate all devices in realtime on top of the CAD layout which was well known to the controllers of the physical system.*

Requirement 7.6 *All simulated devices should be controllable at runtime through a graphical user interface provided by the emulation model. This implies clicking on a simulated device and stopping or resuming its operation, creating failures, pressing buttons, etc.*

7.2 Conceptualization

According to Banks (1998) conceptualization implies the abstraction of a real system using a conceptual model, i.e. a series of mathematical and logical relations between objects. This conceptual model underlies both the DSOL specification

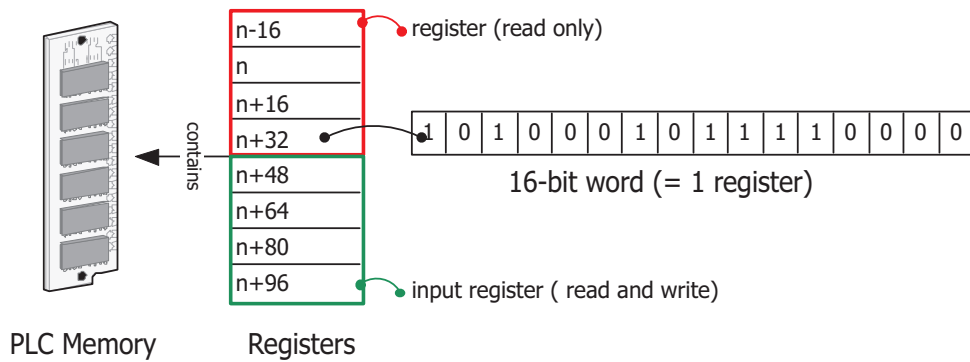


Fig. 7.4: The memory of a programmable logic controller

presented in section 7.3 and TBA’s eM-Plant specification. In this section a clear distinction is made between the *control system* and the *controlled system*. The control system, i.e. the PLC controls the controlled system, i.e. the simulated devices in the emulation model. We start in section 7.2.1 with a conceptual model of the control system, i.e. the PLC. We continue in section 7.2.2 with the conceptual model of the controlled system, i.e. the DSOL emulation model. In section 7.2.3 we discuss the communication between the emulation environment and the real time PLC, and we conclude this section with conceptual models of the overall architecture.

7.2.1 The conceptual model of the control system

To understand how the control system functions, it is necessary to reiterate what we stated in the introduction of this chapter, i.e. that the inner works of a programmable logic control are based on changing bit values of internal memory addresses. This is illustrated in figure 7.4. The memory of a PLC is divided in a number of *registers*, i.e. data elements containing 16 bits. A clear, but subjective, distinction is made between that part of the memory in which write operations take place, and that part which is read-only for external devices to support the consistency of the PLC. The registers that are read by external devices are called *Registers*, while those registers used for writing are called *InputRegisters*.

Some inputs and outputs use more than one bit to read or write a specific value. For example, consider an automated guided vehicle the speed of which is presented as a double value, i.e. a **double** value in Java. In this specific case, 32-bits are needed to store this speed value, and as such two *InputRegisters* are dedicated to the speed of a specific vehicle.

The memory of a PLC is divided into a number of registers, i.e. data elements containing 16 bits.

While *input registers* are registers used for writing data from a (simulated) device into the PLC, *registers* are read-only registers used for reading data.

7.2.2 The conceptual model of the controlled system

The conceptual model of the DSOL emulation model, i.e. the controlled system is presented, in this section. We introduce three types of devices (see figure 7.5) to illustrate the emulation model.

- *Input devices* which *only send* data to the programmable logic controller. Examples of this type include a sensor, an emergency button and a GPS-device.
- *Output devices* which *only receive* data from the programmable logic control. Examples of this type include a lamp and a siren.
- *Combined devices* which *both send and receive* data to the programmable logic controller. Examples include a motor, the operation of which is controlled (or received), and which might send emergency events.

A number of remarks must be made with respect to the conceptual diagram presented in figure 7.5. One, the terminology is rather confusing. Although the name **Interface** might suggest that a Java interface is meant, it represents an abstract class specifying the interface between a simulated device and a PLC. Two, a $0..N$ associative relation is presented between an interface and a specific device. This illustrates the fact that several simulated devices may share the same interface. Three, we see that combined devices are associated with both input and output interface. Four, the interface in figure 7.5 is associated with the PLC. Having introduced conceptual models of the control system and the controlled system, a more detailed conceptual model of the communication between the control system and the controlled system is presented in the following section.

7.2.3 The model-PLC interface

The emulation model must conform to the PLCs industrial *Modbus* communication protocol to ensure that no modification has to be made on the PLC. Modbus is an application layer messaging protocol that provides client/server communication between devices connected on different types of buses or networks (Modicon, 1996). Modbus can furthermore be accessed over a reserved system port 502 on the TCP/IP stack.

Modbus offers services specified by function codes, which are elements of request/reply protocol data units (Modicon, 1996). A *Master-Slave* concept is applied in the field of programmable logic controllers to govern the lower level communication behavior on a network using a shared signal cable (Modicon, 1996). This concept is presented in figure 7.6; and it shows somewhat counter-intuitive terminology.

The emulation model must conform to the PLCs industrial Modbus communication protocol.

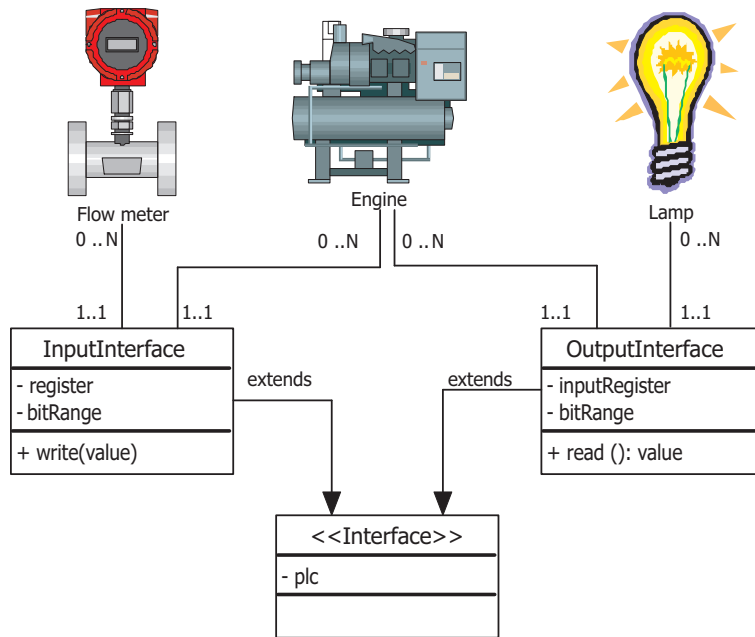


Fig. 7.5: A conceptual model of devices.

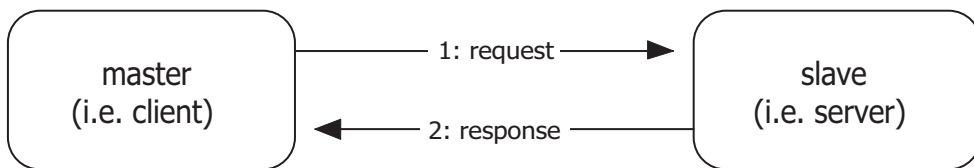






Fig. 7.6: The Modbus master-slave concept

Tab. 7.1: Basic Modbus functions (Modicon, 1996)

Name	Type	Access	Visual representation
discrete input	single bit	read-only	
discrete output, i.e. coil	single bit	read-write	
input register	16-bit word	read-only	
output register	16-bit word	read-write	

A number of basic public functions have been developed in the Modbus protocol for exchanging data that is typical for the field of automation. These functions are presented in table 7.2.3.

Any TCP/IP based implementation of the protocol should extend the protocol data units with an IP specific header. As we will show in the next section, a detailed, reliable and above all high performance implementation of this protocol is crucial for using emulation for PLC testing purposes successfully.

The conceptual architecture for the emulation model is presented in figure 7.7. On the left, the programmable logic controller (PLC) is attached to the TCP/IP network. On the server side of the network a Java implementation of the Modbus protocol ensures adequate communication with the PLC. To minimize the amount of communication over the network, this communication library is connected to a *shadow memory* of the PLC which is part of the emulation model. The shadow memory limits communication in the following ways.

- Although all registers are read periodically from the PLC, the shadow memory will only fire value change events whenever *input values*, i.e. values to be written into the memory of the PLC, are actually changed.
- The shadow memory only sends changed input registers values to the PLC.

Emulated components either write to this shadow memory, and thus to the PLC, or are asynchronously subscribed to changes on the memory addresses which control their behavior. A sensor is an example of such writing, i.e. of an input, component, while a crane is an example of a reading, i.e. of an output, component. The behavior of, and interaction between, components in the emulation model is time dependent and this requires a simulator.

A detailed, reliable and above all high performance implementation of this protocol is crucial for using emulation for PLC testing purposes successfully

Emulated components either write to this shadow memory, and thus to the PLC, or are asynchronously subscribed to changes on the memory addresses which control their behavior.

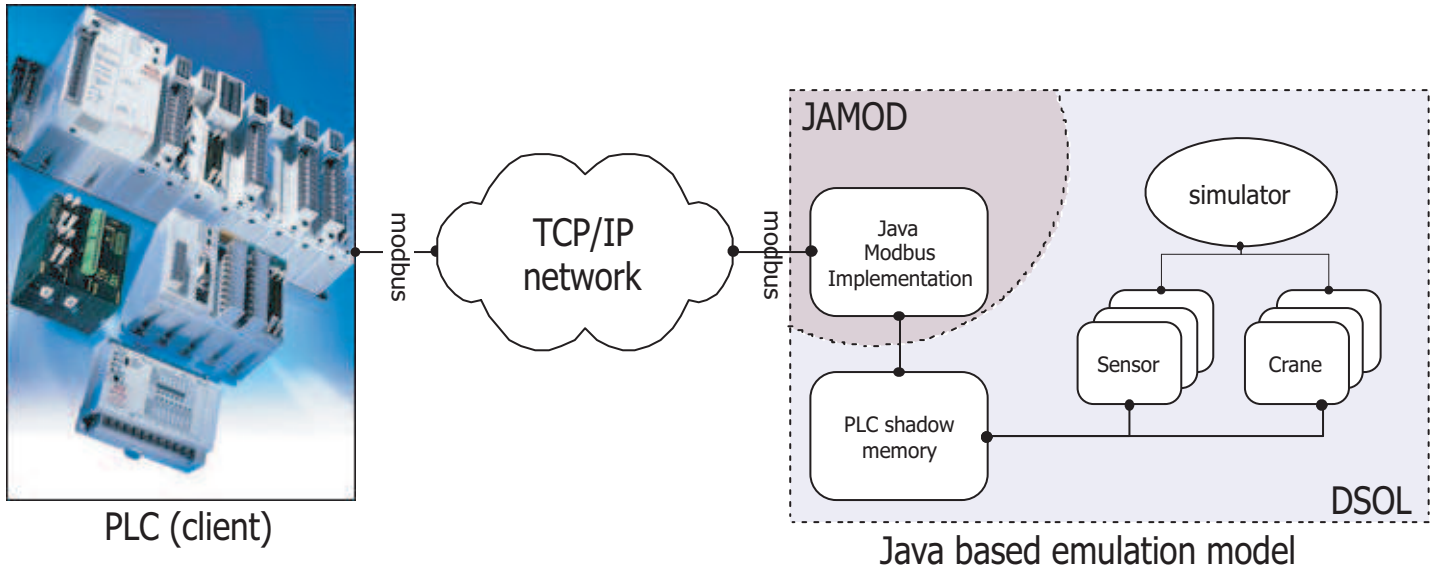


Fig. 7.7: The emulation architecture

7.3 Specification

The next activity in the design of the emulation model was to specify the model in DSOL. We begin in section 7.3.1 with the specification of the Modbus-DSOL communication. We continue with the specification of the simulated devices in DSOL in section 7.3.2.

7.3.1 Modbus communication with DSOL

As argued in the introduction of this thesis, great emphasis is placed on the usefulness of the simulation suite based on the ease with which developers can integrate or communicate with external libraries and services (see requirement 4.2 on page 59).

Although Modbus communication is required for this case to be successful, a Java implementation is clearly not considered among the tasks for a general purpose simulation suite. A Google⁵ search on *Java Modbus* presented an open source implementation of this protocol named *Jamod*⁶. While the development of a dedicated eM-Plant-Modbus library took several (≈ 3) weeks, the availability of this verified, validated and documented library allowed us to achieve DSOL-Modbus communication within hours. This we consider to be a scientific validation of the value of service oriented computing applied to a simulation suite.

The seamless integration between DSOL and Jamod is presented in figure 7.8. The class diagram illustrates how the shadow memory, the input registry and DSOL's event library provide the communication between the PLC and the emulated devices.

The `ModbusMemoryImage` class specifies our implementation of the shadow memory. It implements Jamod's `ProcessImageImplementation` interface which embodies a set of operations for resolving and installing the basic functions presented in table 7.2.3. The interface is implemented by our `ModbusInputRegister` class, which extends DSOL's `EventProducer` class. The `ModbusMemoryImage` further implements DSOL's `EventListener` interface and is asynchronously notified whenever the bit value of an input register is changed. The `ModbusOutputRegister` follows the same structure, but is, because of readability, not presented in this figure.

7.3.2 The specification of emulation components

A class diagram of one of the simulated devices is presented in figure 7.9. For reasons of readability only this simple component, i.e. the sensor, is presented. All other input and output devices are specified in a similar manner.

⁵ <http://www.google.com/search?hl=en&q=Java+Modbus&btnG=Google+Search>

⁶ <http://Jamod.sourceforge.net>

The availability of the off-the-shelf, verified, validated and documented *Jamod* library allowed us to achieve DSOL-Modbus communication within hours.

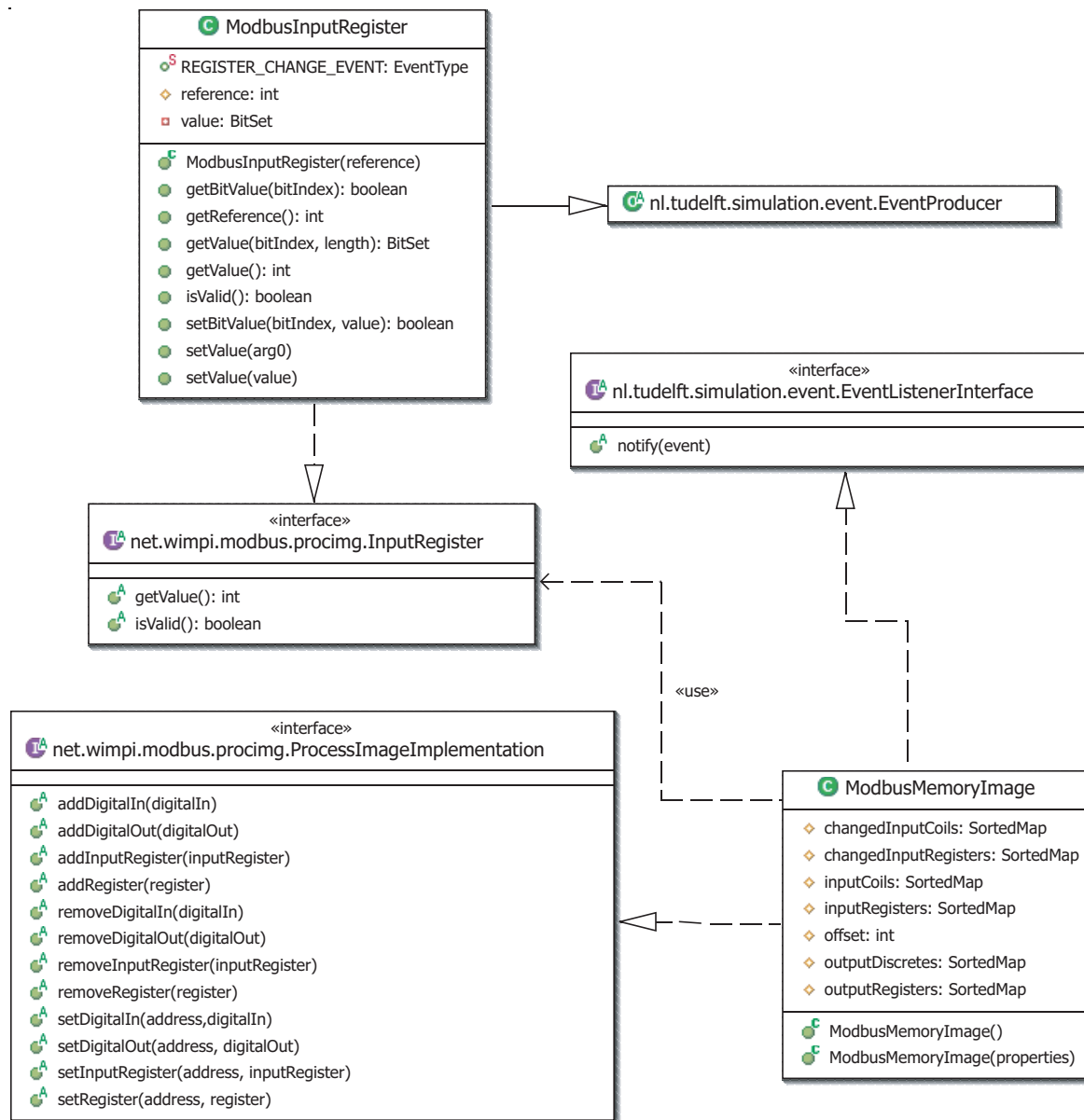


Fig. 7.8: DSOL-Jamod interdependence

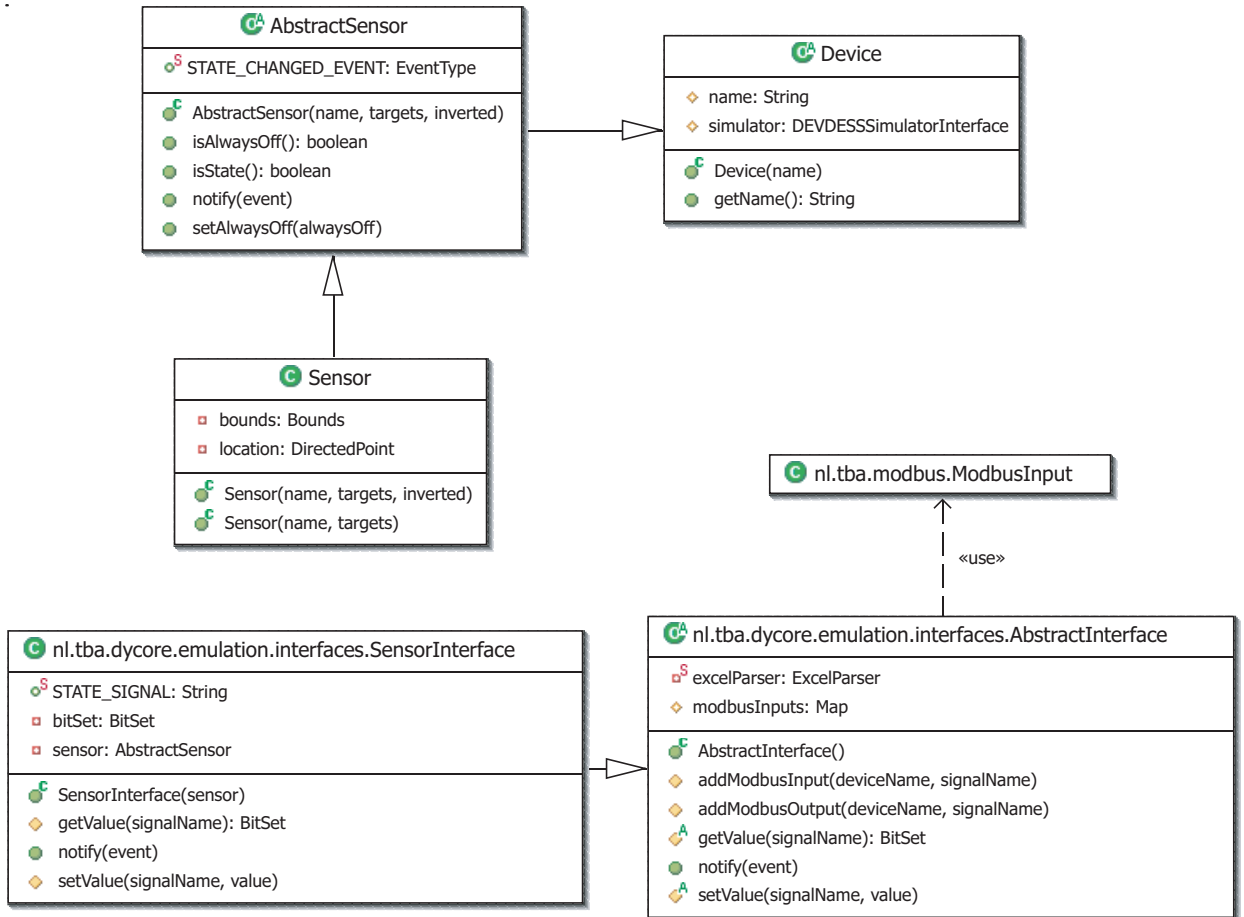


Fig. 7.9: The Sensor emulation component

The `Sensor` class extends the `Device` class. The fact that a sensor has no embedded knowledge of any communication protocol is illustrated in figure 7.9. Since only semantic operations, e.g. the `isState()` operation, are implemented, this simulated device class could well be included in a more general purpose simulation library.

Specific bitwise Modbus memory updates are accomplished by the `SensorInterface` class. This class is asynchronously subscribed to state changes of the sensor to ensure a loosely coupled, efficient communication protocol. The principle of inheritance (see section 3.4.5 on page 42) is applied with the creation of the `AbstractInterface` class. This abstract class uses the *ModbusInput* shown in figure 7.8.

We expected a more elegant simulation model due to our ability to use Java's 3D library. Instead of defining hard coded relations which would inevitably result in emulated objects moving over predefined tracks, the DSOL model uses 3-dimensional bounds to see whether objects intersect. To illustrate this functionality, we present the `detect` method of a `Sensor` below:

```
225  /**
226   * detects a locatable object
227   *
228   * @return whether the sensor has detected a Locatable
229   */
230  private boolean detect()
231  {
232      try
233      {
234          //We compute the space we can currently oversee.
235          Bounds view = BoundsUtil.transform(this.getBounds(), this
236              .getLocation());
237          for (Iterator i = this.targets().iterator(); i.hasNext();)
238          {
239              LocatableInterface locatable = (LocatableInterface) i.next();
240              if (view.intersect(BoundsUtil.transform(locatable.getBounds(),
241                  locatable.getLocation())))
242              {
243                  //Our view intersects with this target.
244                  return true;
245              }
246          }
247      } catch (Exception exception)
```

Instead of defining hard coded relations which would inevitably result in emulated objects moving over predefined tracks, the DSOL model uses 3-dimensional bounds to see whether objects intersect.

```

248     {
249         Logger.warning(this, "detect", exception);
250     }
251     //Nothing detected
252     return false;
253 }

```

The current **view**, i.e. the volume representing the range of sight, of the potentially moving sensor is computed in line 235. This **view** is an instance of **Bounds**, which defines a convex, closed volume that is used for various intersection and culling operations. In line 240 the **intersects** method is invoked on this **view**. The **targets**, e.g. pallets or cars, are provided as an argument; these **targets** are resolved from a **context** (specified by the naming service described on page 68).

7.3.3 The specification of the DSOL-PLC communication

Schludermann et al. (2000) discuss time constraints in an emulation model. These constraints impose great challenges on the simulation environment and on the underlying operation system. To understand these challenges we illustrate the activity sequence of DSOL's simulator, i.e. the realtime clock.

Every time period the simulator sends the input part of the shadow memory to the PLC over the Modbus protocol. Then the simulator reads the output memory from the PLC and notifies subscribed listeners, i.e. Modbus outputs, in case the values have changed. These Modbus outputs block the simulator thread while they invoke mapped semantic operations on the device they control, e.g. `crane.stop()`. After completing this notification, the simulator fires an update animation event, and the CAD based graphical user interface will be redrawn.

The DSOL emulation model was designed with two high-priority threads, i.e. the simulator thread and the communication thread, and one low priority animation thread.

```

-----%
Tally: Backlog of ModbusConnectionThread          %
N=8840 MAX=305.0 MIN=-35.0 AVG=-29.9495475113 STD=16.00013964397 %
-----%

```

Fig. 7.10: The measured backlog (milliseconds) in the DSOL-PLC communication

The reinforcement welding equipment at Dycore requires a maximum time period of $30 \cdot 10^{-3}$ seconds. For the emulation to succeed, DSOL's emulation framework must guarantee the above sequence is completed within this period. The DSOL emulation model therefore was designed with two high-priority threads, i.e. the simulator thread and the communication thread, and one low priority animation

thread. The loosely coupled relation between component behavior and animation furthermore ensured that while the refresh rate of the model was related to the period of the PLC (≈ 35 Hz), the refresh rate of the animation could be slower (≈ 5 Hz). The value of this distinction is that the increased priority of the communication thread allowed the required period of the emulation model to be reached (see figure 7.10). Although the values presented in figure 7.10 show that DSOL was in general well able to communicate every $30 \cdot 10^{-3}$ seconds with the PLC, there were occasions when a positive backlog occurred.

Because of Java's ability to spread tasks over multiple threads and to differentiate their priorities, DSOL outperformed eM-Plant by a wide margin in this task⁷. This achievement supports the value of interface based design. The openness of the simulation suite invited us to design a specific, performance aware `RealTimeClock` class which implements the `SimulatorInterface`.

7.3.4 Experimentation

We present the graphical user interface of the Dycore emulation model in figure 7.11. In this user interface the underlying Autocad files are rendered by an external, open source, geographical information system service named *Gisbeans* (Jacobs and Jacobs, 2004). We consider the ease of using this external service, which is in no way related to the domain of simulation, to again act as a validation for the service oriented paradigm (see principle 3.2 on page 37). The popup screen titled H2031 illustrates how devices can be individually and independently controlled (fulfilled by the introspection service described on page 69).

Although the values presented in figure 7.10 show that DSOL was in general well able to communicate every $30 \cdot 10^{-3}$ seconds with the PLC, there were occasions when a positive backlog occurred.

7.4 Conclusions

The most important question yet to be answered is to what extent this case supports our hypothesis that a service based simulation suite provides more effective decision support. To answer this question we need to clarify the differences between the two specifications and link them to our $N_n-N_m-N_o$ approach towards the design of decision support systems.

The first appearance of the $N_n-N_m-N_o$ paradigm underlying DSOL is the seamless integration of several off the shelf libraries and services. Poi⁸ was used to read/write Microsoft Excel files, Gisbeans was used to render CAD files and Jamod was used to communicate over the Modbus protocol. This is in clear contrast with the eM-Plant implementation developed by TBA. The communication between

⁷ Although we have not been given detailed information on the performance of TBA's eM-Plant model, the claim of DSOL substantially outperforming eM-Plant has been made by TBA's engineers.

⁸ <http://jakarta.apache.org/poi/hssf/>

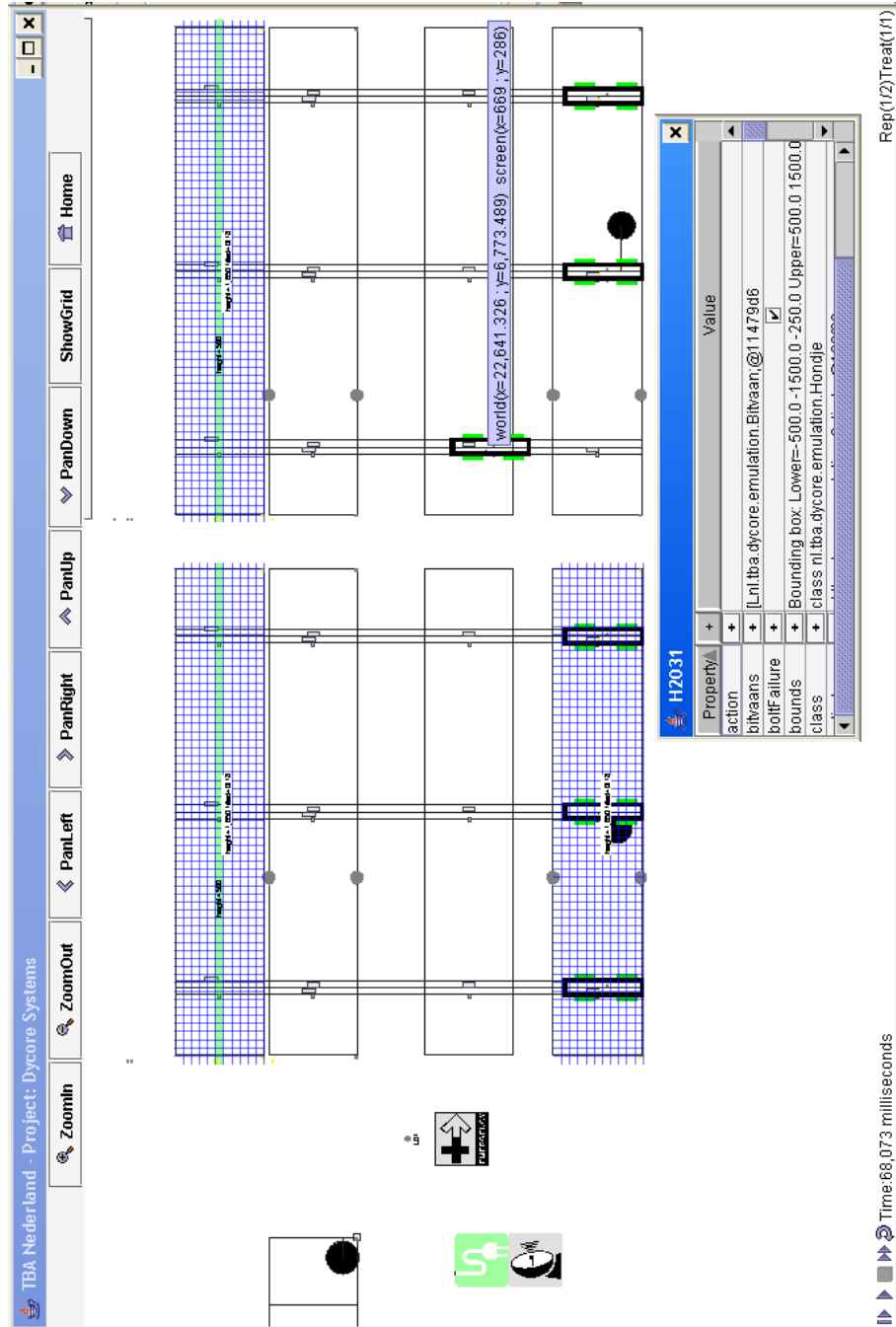


Fig. 7.11: The graphical user interface

the eM-Plant emulation model and the PLC required dedicated, proprietary engineering. We have shown that the service oriented architecture underlying DSOL enabled us to select required services, and thus to tailor the suite for this specific case study. We conclude we have provided scientific evidence to support the value of service oriented architecture for the effectiveness of DSOL as a decision support system.

The openness of the DSOL platform proven to be crucial for meeting the performance constraints of the case. The fact that tasks can easily be dispersed over several threads, each with a specific priority meant that the performance of DSOL's realtime clock outperforms eM-Plant substantially.

The loosely coupled relation between component behavior and animation ensured that while the refresh rate of the model was related to the period of the PLC (≈ 35 Hz), the refresh rate of the animation could be slower (≈ 5 Hz)⁹. The value of this distinction is that through the increased priority of the communication thread, the required period of the emulation model could be reached (see figure 7.10). Although the values presented in figure 7.10 show that DSOL was in general well able to communicate every $30 \cdot 10^{-3}$ seconds with the PLC, there were occasions where a positive backlog occurred. This justifies further research on the applicability of DSOL on a real-time operating system. We conclude that this case study has provided evidence for a loosely coupled structure between the model and its environment; we furthermore conclude that this was essential for the usefulness of DSOL in the context of emulation.

The inclusion of Java's 3D model resulted in a track-less infrastructure model. The sensors, conveyors and cranes can actually scan their neighborhood for pallets to be moved or lifted. Such a three-dimensional library is not available in the eM-Plant specification, which results in more constraining, dedicated relations between infrastructural objects if eM-Plant is used.

As these differences show, the $N_n-N_m-N_o$ paradigm underlying DSOL especially seems to show its added value whenever models require sophisticated domain specific challenges. While traditional simulation environments are designed for a one purpose, one formalism and one target audience, this case clearly shows the added value to be gained from an interacting, open simulation suite.

The $N_n-N_m-N_o$ paradigm underlying DSOL especially seems to show its added value whenever models require sophisticated domain specific challenges.

⁹ These measurements are conducted on a 1Ghz Pentium III, 512MB RAM system with DSOL 1.6 on JRE 1.4.2.

8. CASE: FLIGHT SCHEDULING AT KLM

The development of a flight plan acceptance model for the Royal Dutch Airlines, KLM, is presented in this chapter. Like the case study presented in chapter 7, the case study presented in this chapter serves as an instrument in the validation of DSOL with respect to its effectiveness for simulation. The focus of this case study is different though; where the focus of the Dycore case study was on the usefulness of DSOL, improved usability and usage form the primary goals of this case study. The content of the case, i.e. the plan acceptance process, is presented in section 8.1.1. We present the challenges and requirements of the case in section 8.1.2 and illustrate the value of this case for our research. A conceptual model of the delivered simulation model is presented in section 8.2. We end this chapter with conclusions on the validity of DSOL with respect to its usefulness, usability and usage. Note, this chapter has been read and its content has been endorsed by the Decision Support and Operations Control departments of KLM. The content of this chapter reflects the paper '*Flight scheduling at KLM*' (Jacobs et al., 2005a)

The development of a flight acceptance model for the Royal Dutch Airlines, KLM, is presented in this chapter.

8.1 Introduction

Royal Dutch Airlines, or KLM, is the Netherlands' largest airliner and as such one of the largest in Europe; it was founded in 1919 and since then it has experienced constant growth. In the fiscal year 2003/2004, KLM employed 34.529 people and maintained a fleet of 188 aircraft (KLM, 2004). KLM executes a schedule connecting over 400 cities in 85 countries on 6 continents. An estimated 23.4 million passengers and some 529,000 tons of freight were transported over its network in 2003/2004 (KLM, 2004). KLM has recently merged with Air France; the new airline is the biggest in Europe and number three in the world.

KLM adapts its flight schedule at least four times per year to accommodate changes in demand. Schedules are developed by the Network department, a business unit with a strong commercial focus, whose main interest is to maximize profit by maximizing the number of passengers, flights and flight connections. Within KLM, Operations Control is the business unit responsible for operating the schedule. Its interest lies in executing a feasible schedule which implies less passengers and fewer flights. A sub-department of Operations Control, called Plan Acceptance

KLM adapts its flight schedule at least four times per year to accommodate changes in demand.

A department called *Plan Acceptance Management* assesses the feasibility of a new schedule supported by a simulation model.

This simulation model, delivered in 2003, and specified in Arena, is called *OPiuM*.

Management, assesses the feasibility of a new schedule, and reports to Operations Control on whether to accept it. Due to the often conflicting interests of Network and Operations Control, objectivity and rationality in the plan acceptance process are highly valued. To achieve such rationality, the decision support department of KLM was requested to develop a simulation model that would support gaining insight into the operational consequences of a new flight schedule. This simulation model, delivered in 2003, and specified in Arena, is called *OPiuM*, and is used ever since.

8.1.1 The plan acceptance process

Every quarter, Network develops a new initial plan based on commercial and strategic insights. Two months before the start of a new schedule, the plan is presented to Operations Control. Plan Acceptance Management then reviews the schedule and evaluates resulting agreements with capacity service providers, e.g. cabin crew, pilots, and the engineering divisions. The combination of a schedule and matching agreements is called the operational plan, or OP. An example of such an operational plan is presented in figure 8.1.

Operations Control and Network have a service level agreement that describes the performance that Operations Control must deliver.

Operations Control and Network have a service level agreement that describes the performance that Operations Control must deliver. The three key performance indicators of this agreement are the completion factor, i.e. the percentage of flights executed, the aggregated arrival delay and the aggregated departure delay, expressed as a percentage.

From the moment Operations Control accepts a plan, it is considered to be its owner; specific aircrafts are now to be assigned to individual flights. Any further adaptations to the plan have to be evaluated and approved by Front Office, which is the virtual department managing operations on the day of execution. This department is virtual since it results from co-operation between all involved parties at the day of operation. The work of Front Office consists of minimizing effects of daily problems on succeeding flights. Two weeks before the beginning of the operational plan, passenger bookings are matched with aircraft capacities.

The three key performance indicators of this agreement are the completion factor, the aggregated arrival delay and the aggregated departure delay.

To improve operational insight in future operational plans, Decision Support developed a simulation model, named *OPiuM*¹. This model reflects KLM's philosophy of evaluating individual business units based on their performance in the execution of sub-processes. The process of an individual plane is divided into four sub-processes, which are referred to as *process building blocks* and are presented in figure 8.2. A department responsible for the execution of a sub-process is called a capacity service provider. KLM's management made capacity service providers individually responsible for measuring and publishing service time distributions.

¹ *OPiuM* is a Dutch abbreviation for Operational Performance (in uitvoering) Model.

All flights of a schedule are simulated in OPiuM. Disturbances in OPiuM are the result of the difference between a value drawn from a statistical distribution and an planned average process time.

All flights of a schedule are simulated in OPiuM. Disturbances in OPiuM are the result of the difference between a value drawn from a statistical distribution and a planned average process time. The statistical distributions used in OPiuM are based on the service time distributions, provided by the capacity service providers. Whenever there is a disturbance in the simulated schedule, OPiuM optimizes the remainder of the schedule by evaluating a number of potential measures. Such measures include accepting the disturbance, swapping planes and canceling the flight. Penalties, awarded to all these measures, are used to evaluate the quality of the remainder of the schedule.



Fig. 8.2: KLM's process building blocks

Although the current specification of OPiuM in Arena is considered to be a great success, managers and employees must be constantly warned not to overtrust the outcome of the model, KLM believed there was added value to be obtained from a renewed specification of the OPiuM model in DSOL.

8.1.2 Why a renewed specification in DSOL?

The challenges noticed by KLM with respect to the usefulness, usability and usage of their current specification of OPiuM forms the topic of this section.

There were several reasons for a renewed specification of OPiuM in DSOL all of which reflect the need for an open, distributed simulation suite.

- One of the most noticeable problems was that while the optimizer requires operations research strategies to deliver the required output, Arena is clearly not designed for such algorithmic specifications. As a result OPiuM is mostly specified in Microsoft Visual Basic² and Arena is, besides providing an event calendar and thus the simulated time, almost circumvented as a simulation language.
- Another problem is Arena's model execution environment. Although OPiuM was developed by highly experienced simulation experts, the users of OPiuM,

² Visual Basic is a trademark of Microsoft Corporation (<http://msdn.microsoft.com/vbasic/>)

i.e. employees of Plan Acceptance Management, are not highly educated in this area, and while they are experts in the domain of flight scheduling, the execution and development environment of Arena is considered to be too complex and as such lacks usability qualities for those that must use it.

- Due to Arena being specialized software at KLM, and because of the complicated structure and deployment of Arena licenses, in addition to the specific versions of the libraries used to accomplish the interaction between Arena and Microsoft Visual Basic, the Decision Support department developed OPiuM and by default became responsible for its installation, maintenance and service for end-users.
- Arena does not provide any support for the collaborative model specification, nor does it provide support for distributed, concurrent model execution.
- Arena is difficult to integrate with external distributed data sources. Since capacity service providers must individually publish service times, support for integrating information would be very much appreciated.
- An overall conclusion with respect to the current specification in Arena was that KLM foresees that usage of OPiuM may well shift to a more operational mode, in which daily problems are evaluated, if the current specification has not reached its limits with respect to scalability and performance.

Given the above, KLM and Delft University started to specify OPiuM in DSOL. The value for this case for the validation of DSOL is twofold. One, the case has also been specified in a traditional, alternative, simulation language; it offered a possibility to compare the two. Two, the decision support department at KLM initiated contact and proposed the case; we did not seek it out as a validation case. We wish to emphasize the fact that KLM was not involved in the design phases of DSOL; and we had no prior contact before the start of the case study.

8.2 *Conceptualization*

We will now introduce the conceptual diagrams of the case, to provide insight into the processes of executing a flight schedule. A schedule consists of fleetlines; a sequence of flights scheduled during the schedule period to be performed by one aircraft. If a flight is delayed, Operations Control can take several measures to optimize the remainder of the schedule. To evaluate the consequences of a potential measure, a value function is assigned to each measure, expressing a sanction, in minutes delay, for a particular flight, at a particular moment in time. The following measures are used in the model.

If a flight is delayed, Operations Control can take the following measures to optimize the remainder of the schedule: swapping two fleetlines,...

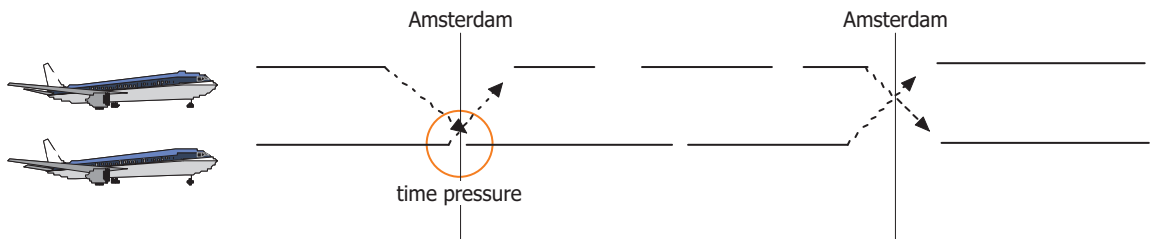


Fig. 8.3: The swap measure

- *Swapping two fleetlines*: one of the measures is to swap two fleetlines. This implies that the scheduled flights for a particular plane for the remainder of the rotation are swapped with those scheduled for an alternative plane. A rotation is the sequence of flights from Schiphol to Schiphol. Usually this involves two flights, or legs, but sometimes it involves three or more. The swap measure is illustrated in figure 8.3. If a delay causes time pressure, and thus an inevitable delay, on the next flight, swapping a rotation between two planes may well be a rational measure to take. Swaps are only performed at Schiphol airport and preferably between two planes of the same sub-type. This is called a *registration swap*. Only if such plane is not available, are planes of another type considered. The sanctions for this measure are expressed in minutes and based on the differences in passenger capacity of the aircrafts involved. There is a base sanction plus a leg-sanction for every swapped leg, except for registration swaps, where only the base sanction applies.

...using a reserved flight,...

- *Using a reserved flight*: this is an aircraft that is kept idle for a longer period of time, i.e. several hours to days. A reserved flight is used to schedule a standby aircraft and crew, which are only used whenever a problem in the schedule occurs. When a flight must be executed but the originally assigned aircraft is unavailable a reserve aircraft may be used. The sanction and value function of a reserve are equal to the swap measure.

...reducing the maintenance time of a plane and...

- *Reducing maintenance time*: maintenance can be shortened by approximately 15% of the regular maintenance time by increasing the amount of assigned resources, i.e. engineering staff and equipment. The sanction for this measure depends on the type of airplane.

...canceling a flight.

- *Canceling a flight*: a cancel-measure implies that an entire rotation is canceled, i.e. it will not be executed. The sanction is evaluated per leg and is very high, which is not surprising since cancellation is the most radical solution.

Operations Control is responsible for the execution of flights and thus for the optimization of the schedule. A sequence diagram of this `executeFlight` operation is presented in figure 8.4.

The following characteristics of this sequence diagram are worth discussing. One, on the execution of a flight, an external optimizer is requested to optimize the schedule. Two, this optimizer is defined as an interface. By using an interface, we adhere to the principle of design by contract (see principle 3.4.9 on page 44) and as such emphasize on a loosely coupled relation, i.e. we place the emphasis on the replaceability of a particular implementation. A domain specific, proprietary, external service can easily be used to do the actual optimization. Three, the actual execution is implemented in the `Aircraft` which adheres to the principle of separation of concerns (see principle 3.2 on page 37).

Although the use of an external, well validated optimizer is encouraged, a reference implementation of the interface is presented in figure 8.5. The following characteristics of this implementation are worth discussing. One, the optimizer has no relation with the DSOL simulator. This ensures that the algorithm used to optimize a flight schedule can be shared between the simulation model, i.e. `OPiuM`, and the information systems supporting daily flight execution. The potential use of a domain specific optimizer is a direct result of the fulfillment of requirement 4.2 on page 59. It furthermore opens the door to use `OPiuM` in a more operational daily environment. Two, measures follow a transaction model, which supports an initial try and a final perform. Three, both value functions assessing the consequences of individual measures and the reference time providers, i.e. `NormTimeProvider` are interfaces that emphasize on the replaceability of a particular implementations.

The actual execution of a flight is the responsibility of an aircraft, see the sequence diagram of figure 8.4. The `Aircraft` class is presented in figure 8.6. The following characteristics are presented in this figure. One, static parsing methods in both the `Aircraft` and the `SubType` directly parse the *Flash*³ schedule presented in figure 8.1. Two, the `State` class represents the states reflecting the sub-processes of an plane. The sequence is specified in the `nextState` operation.

KLM's schedule is conceptualized in figure 8.7. The `Fleetline` holds a number of *FlightLists* each holding a number of *Flights*. DSOL's event package enables flights to fire events whenever a delay is incurred, and to compute the performance indicators of a schedule; DSOL's statistical objects are asynchronously subscribed to these events.

Although the use of an external, well validated optimizer is encouraged, a reference implementation of the interface is presented in figure 8.5.

³ *Flash* is the name of the KLM proprietary software used to create flight schedules.

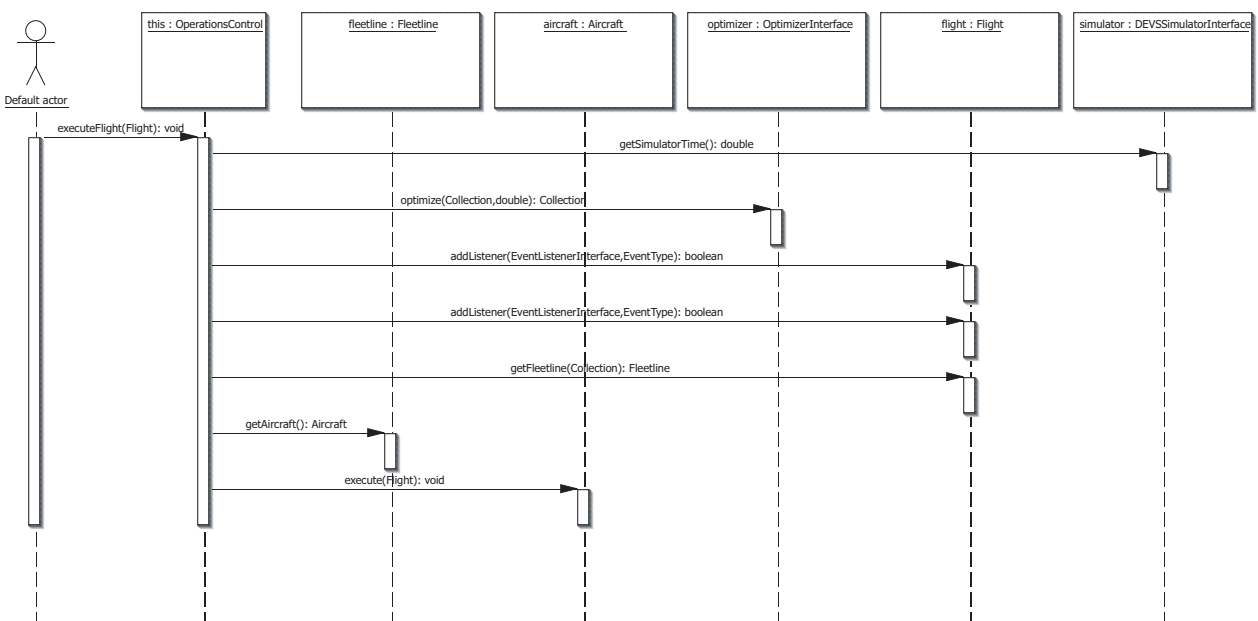


Fig. 8.4: Sequence diagram of the executeFlight operation

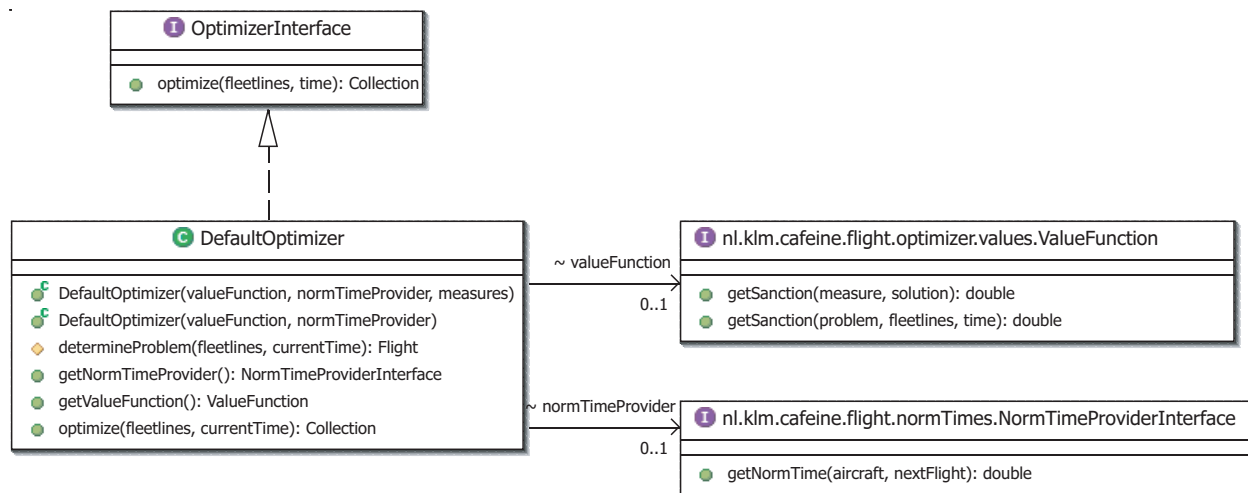


Fig. 8.5: A reference implementation of the `OptimizerInterface`

8.3 Specification

We present the specification of the OPiuM model in DSOL in this section. The actual values of all processes and resources can be found in van Duin (2005); we introduce only those aspects that are relevant for understanding the extent to which the DSOL specification of OPiuM differs from the prior OPiuM model specification in Arena.

8.3.1 Input specification

An example of an operational plan is illustrated in figure 8.1 with the screenshot of the KLM proprietary schedule software named *Flash*. DSOL supports input and output of this file format to ensure that the usability of OPiuM and Flash for the end-user is not affected. This validates the value of requirement 4.2 on page 59. Although capacity service providers publish service times individually and autonomously, Decision Support is currently obliged to download such information and to export it into Microsoft Access⁴ files which can be read by Arena. The DSOL specification of OPiuM supports both the Microsoft Access, and any remote text-based proprietary document format. As a consequence updated information can be automatically downloaded over ftp, http or nfs. Java’s JDBC⁵ and

DSOL supports input and output of the KLM proprietary schedule software named *Flash* (see figure 8.1)

⁴ Microsoft Access is a trademark of Microsoft Corporation (<http://www.microsoft.com/>)

⁵ JDBC technology is an application program interface that provides cross-database management system connectivity to a wide range of SQL databases.

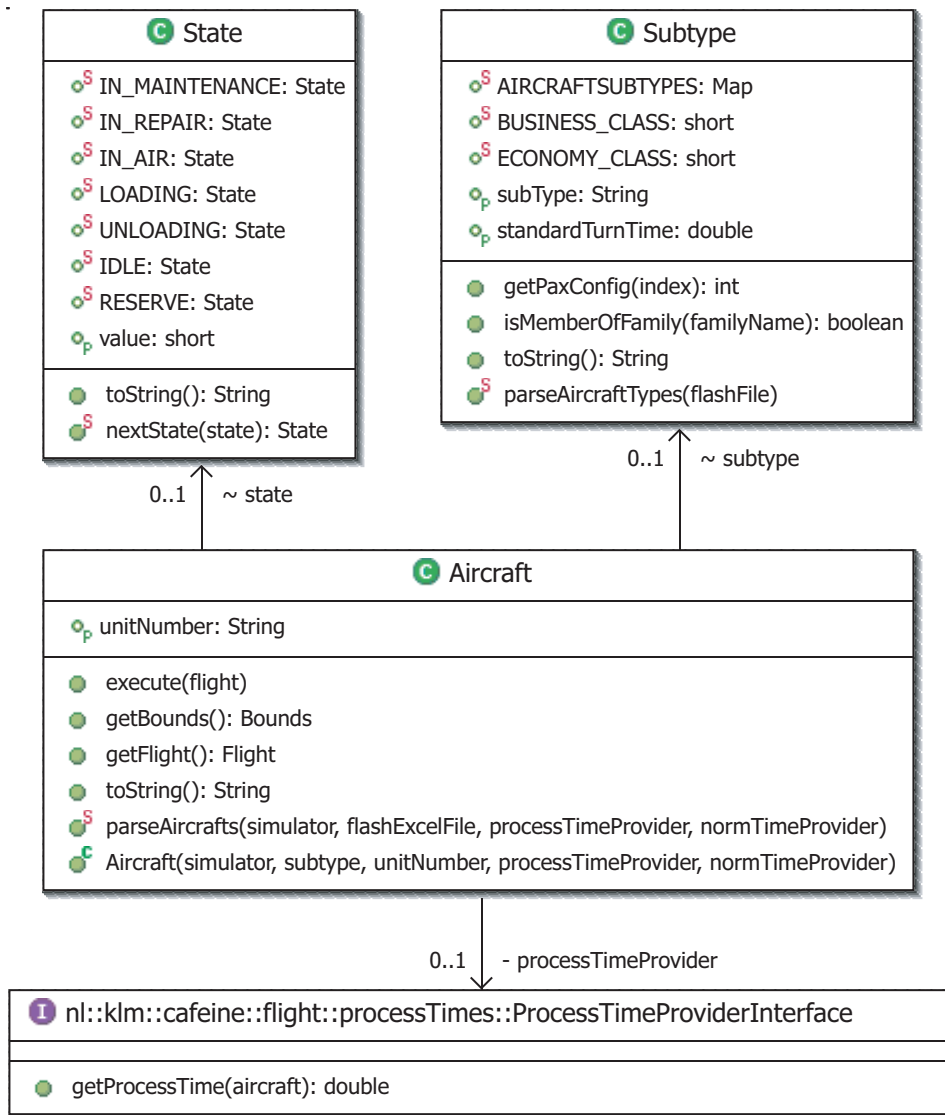


Fig. 8.6: Class diagram of Aircraft

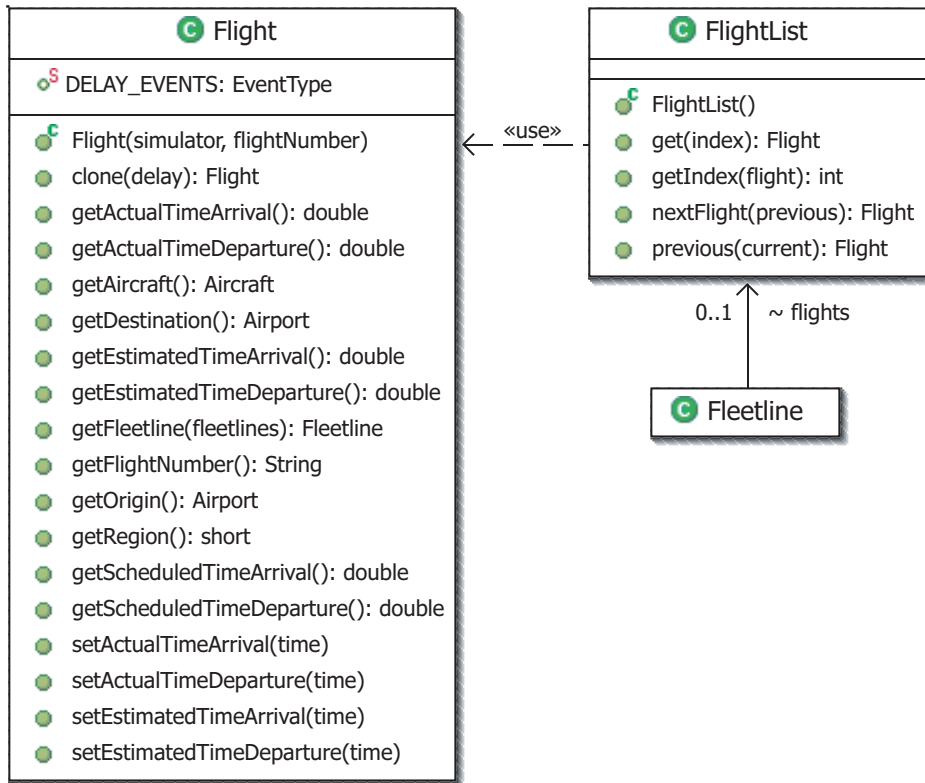


Fig. 8.7: Class diagram of Fleetline

```

<model>
  <model-class>nl.klm.cafeine.model.Model</model-class>
  <class-path>
    <jar-file>http://www.simulation.tudelft.nl/airfields.jar</jar-file>
    <jar-file>ftp://anonymous:klm@ftp.klm.com/pub/data.jar</jar-file>
    <jar-file>file:/C:/development/caffeine/world.jar</jar-file>
  </class-path>
</model>
<properties>
  <property key="DATABASE_PROPERTIES" value="/database.properties"/>
</properties>

```

Fig. 8.8: Experimentation in the DSOL specification of OPiuM

JNDI⁶ standards further result in support for remote relational and directory-based databases. This support is illustrated in the model definition of the experiment definition presented in figure 8.8. The connection arguments for connecting to a remote database, e.g. Microsoft Access or Oracle, are supplied in the text based `database.properties` file. DSOL's support for such wide range of potential data sources validates the value expressed in figure 5.1 on page 64.

8.3.2 Output specification

The DSOL specification of OPiuM supports a number of output modes. Besides charts and Microsoft Excel files, the DSOL specification of OPiuM supports animation on top of a geographical information system, i.e. Gisbeans (Jacobs and Jacobs, 2004). Several characteristics of DSOL's animation capabilities are presented in figure 8.9.

- DSOL supports multiple, remote animation screens which are concurrently subscribed to the simulation model. This validates the value of DSOL's strong support for remote asynchronous communication, which is provided by event service presented in section 5.5 on page 70. It furthermore validates the value of a loosely coupled, pull-based approach to animation (see section 5.9 on page 95).
- The geographical information system, Gisbeans, supports layered based rendering; detailed information is only shown at detailed zoom levels.

DSOL supports multiple, remote animation screens which are concurrently subscribed to the simulation model.

⁶ Java Naming and Directory Services

- The animation of the planes is state dependent, i.e. the actual delay is represented under the plane.
- Clicking on a specific plane enables users to drill down and actually to get operational insight into the status of the plane, e.g. the position and the estimated delay (see the introspection service on page 69).

Clicking on a specific plane enables users to drill down and get operational insight into the status of the plane, e.g. the position and the estimated delay.

8.4 Conclusions

We will now evaluate the current specification and the extent to which we have accomplished the requirements presented in section 8.1.2. The first requirement was to overcome the circumvention of the simulation language to specify the algorithms of schedule optimization. Without further elaboration, we may well conclude that using the Java programming language for the specification of the model fulfills this requirement. We argue that the specification of these algorithms provides evidence for the achievement of a conceptual modeling freedom (see section 4.7); The open architecture of DSOL furthermore prevents a doubtful boundary between parts that are available to designers and parts that are shielded. DSOL for example provides full access to the event list of the discrete simulator.

A second requirement was to distinguish end-users from simulation model builders and to target specific support environments for their tasks. In DSOL, model builders are supported with state-of-the-art software engineering tools such as an integrated development environment, e.g. Eclipse⁷, and a Java project management tool, e.g. Maven⁸. End-users are supported with a web-portalled environment which is tailored for specific usage. We conclude that the separation of model builders and model users has improved the usability of OPium; both are supported in a tailored environment.

A third requirement was to deliver a suite of KLM standard information system services that allowed KLM's IT department to take responsibility for and control installation and end-user support. Since KLM has standardized all in-house developments in the Java programming language, we may well conclude that our specification fulfills this requirement.

The fourth requirement was that the decision support department should be supplied with tools that could meet a need to support collaborative model specification. This case taught KLM how to use a concurrent versioning system to synchronize changes on a central model repository. A more general conclusion for our research is that the KLM case provided a clear justification and validation of

A general over reaching conclusion is that, from our perspective, the DSOL implementation has yet to reach its limitations with respect to scalability and performance.

⁷ Eclipse is a trademark of the Eclipse consortium (<http://www.eclipse.org>)

⁸ <http://maven.apache.org>

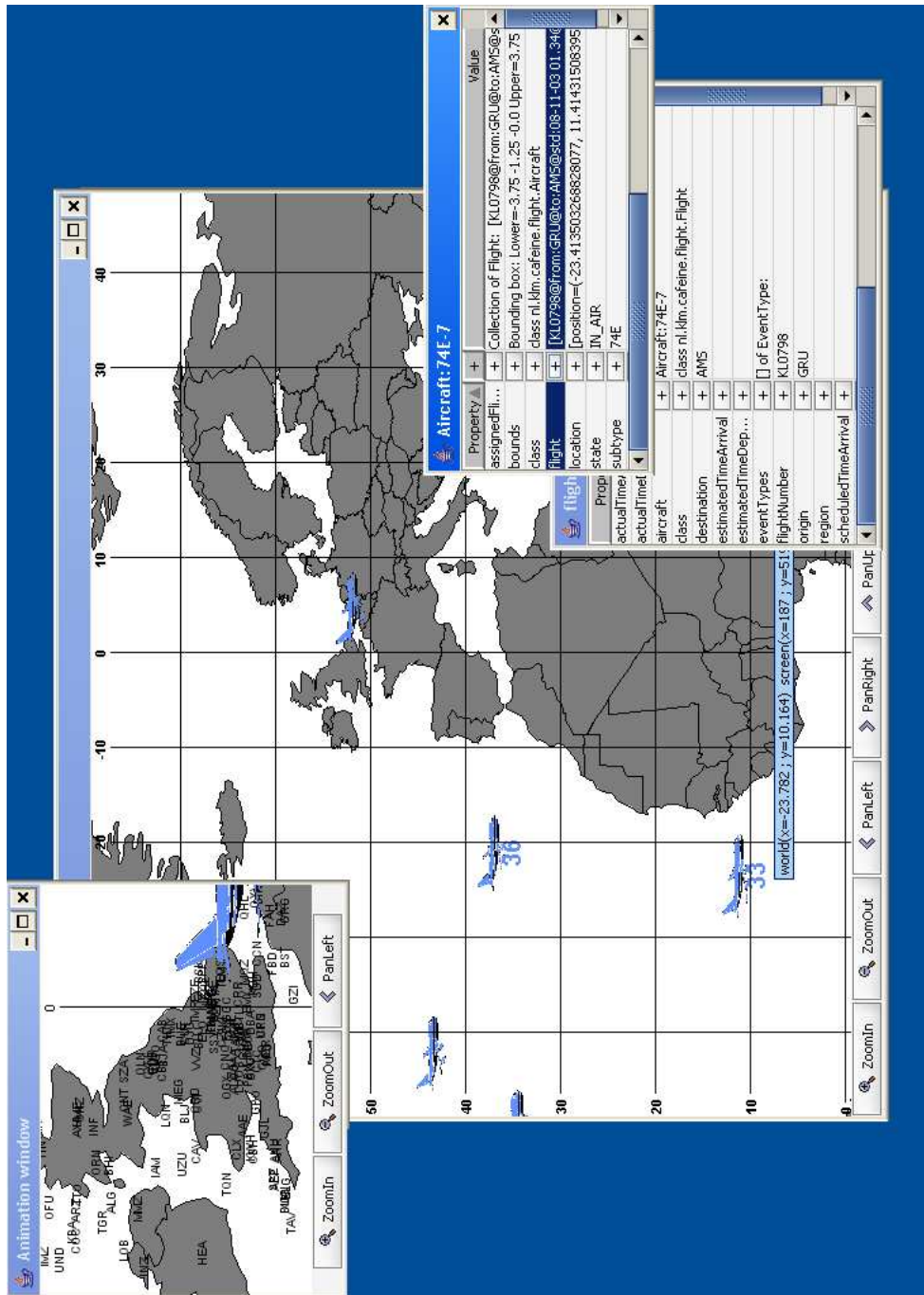


Fig. 8.9: Animation in the DSOL specification of OPiUM

the $n > 1$ decision makers. With this renewed specification multiple decision makers could share their opinion on specific flight schedules, and the underlying value functions and optimization strategies can clearly be used by the actual day-to-day operations.

Java is a general purpose programming language designed for a networked, distributed environment. The DSOL specification of OPiuM makes use of libraries to connect to external databases, in-memory databases, directory services and remote files over any network protocol, e.g. ftp or http. A diagram showing the fulfillment of the requirement to link to external data sources is presented in figure 8.8. We conclude that DSOL improved the usage of OPiuM within KLM.

A general over reaching conclusion is that, from our perspective, the DSOL implementation has yet to reach its limitations with respect to scalability and performance. The possibilities to *not* animate, and to distribute execution over multiple processors, support our strong conviction that the DSOL specification is well equipped to serve more operational, daily processes, however more research is required to support this.

The possibilities to *not* animate, and to distribute execution support our strong conviction that the DSOL specification is well equipped to serve more operational, daily processes.

9. EPILOGUE

In the last chapter of this thesis, we return to our main objective: improving the effectiveness of decision support with a well designed simulation suite for a studio based decision process. We observed that while traditional simulation environments are designed to support one actor, one formalism, one operating system and one model, actors that have to make decisions in ill-structured situations require support systems designed to support concurrently multiple decision makers, who may be dispersed over multiple locations, and who are experimenting with multiple models specified in multiple formalisms. We concluded that we need a new paradigm, or world view, for the design of decision support systems which we named the N_n - N_m - N_o paradigm.

We based our research on the assumption that the effectiveness of decision support is improved in a studio-based approach. We hypothesized that we could design a simulation suite based on an N_n - N_m - N_o paradigm and that this suite would provide more effective decision support.

A review of the research questions is presented in section 9.1. Conclusions regarding the success of embedding DSOL in the simulation community are discussed in section 9.2. We have argued throughout this thesis that the domain of simulation should be grounded on the N_n - N_m - N_o paradigm. In the last section of this thesis we discuss the consequences of this paradigm and present recommendations for further research.

9.1 *A review of research questions*

We review the set of revised research questions, presented in chapter 2, in this section.

Research question 1 *Can we create a simulation suite which takes full advantage of the distributed, service oriented computing paradigm?*

We presented the object-oriented system description as the basis for a service-oriented design in chapter 3. We defined a service as *'the specification of an object-oriented (sub)system that offers a cohesive set of functionality via one or more interfaces'* (see page 63). Several principles (3.4.2, 3.4.3 and 3.4.9 on page 44)

In the last chapter of this thesis we return to our main objective: improving the effectiveness of decision support with a well designed simulation suite.

present the value of *designing by contract* to achieve a service based information system, i.e. a suite. To address the extent to which such service oriented design is accomplished, i.e. our first research question is answered, we present the following evidence:

Usefulness of the decision making process is improved through the current availability of several formalisms and well documented interfaces for domain specific libraries.

- The choice for the Java programming language, that was presented in section 5.2, was made on the basis of its strong bias for distributed, service oriented characteristics. Java supports interface based design, provides distributed deployment of services and web-based communication.
- In section 5.3 we stated that the main reason not to base the development of DSOL on already existing Java based simulation environments was that these environments lack a service oriented paradigm, e.g. objects are not serializable or no interfaces are defined.
- DSOL is presented as a set of services in figure 5.2 on page 67. These services are designed for a specific functionality in the context of simulation. Considerable attention is given to the dependencies between them to ensure the modularity of the suite (see table 5.1 on page 70).
- Strong evidence for the interface based design of DSOL is presented in the class diagram of figure 5.6 on page 75; the absence of classes in this figure emphasizes the decoupled, open and service oriented, i.e. interfaces, philosophy underlying DSOL.

Usability is improved and supported by the distributed web-based access, the separation of the modeling environment and experimentation environment and the introduction of state-of-the-art software engineering tools.

Based on this evidence we conclude that we may answer the first research question positively and that we indeed succeeded in developing a suite of open, interacting services. We have furthermore found strong evidence, in both validating case studies, for the value of service based design with respect to the effectiveness of a decision support system.

We showed in chapter 7 how a realtime clock implementation of DSOL's **Simulator-Interface** enabled the use of DSOL in the time-constant and performance-defiant domain of emulation. We furthermore showed how the suite was linked to other external services (see section 7.3.1). We conclude that it was possible to use DSOL in this specific domain because the DSOL suite could be tailored, that is: specific performance tuned services were deployed in the DSOL suite.

We showed the value of service based model conceptualization and specification in the KLM case study. We were able to specify the optimization algorithms as a service to be used by the simulation model; this loosely coupled relation provided us with the ability to share the optimization service between the OPiuM model and KLM's operational information systems.

Research question 2 *Can we create a simulation suite which supports conceptual modeling freedom using discrete and continuous simulation models?*

To address this research question we recall the formalism transformation graph (figure 4.5 on page 52) and the class diagram presenting the hierarchy of simulators (figure 5.6 on page 75). Since we have specified a simulator for the combined continuous and discrete formalism, i.e. the DEVDESS formalism, we conclude that we may answer this research question positively: all other formalisms can be embedded into this formalism.

We have shown that models may use specific formalisms which are embedded in either the continuous *DESS*, the discrete *DEVS* or the combined *DEVDESS* formalism. An example of such formalism is the process interaction formalism presented in section 5.6.2. The ability to specify such formalism is a results from decoupling the formalism used in the model, e.g. process interaction, and the formalism of the simulator, e.g. discrete event formalism. This ensures that all other formalisms presented in the formalism transformation graph that are not already implemented in DSOL, can be implemented in DSOL.

The value of providing conceptual modeling freedom based on the support of discrete and continuous simulation models for the effectiveness of a decision support system is presented in a number of case studies. In the KLM case study we showed that the optimization algorithms could be directly implemented in DSOL: circumvention of the simulation environment was no longer needed. We conclude that this is a consequence of, and shows the value of supporting modeling freedom. The flexible assembly case (section 6.1.2) and the emulation case (chapter 7) furthermore show how the models were conceptualized and specified in multiple formalism. A more general example is that every example presented in section 6.1 is specified in its own formalism; thus we have clearly shown the value of this conceptual modeling freedom for the domain of simulation.

Research question 3 *Can we create a simulation suite based on a loosely coupled structure between a model and its environment, e.g. animation, statistics, optimization components?*

To answer this question we refer to the presentation of DSOL's animation framework given in chapter 5. DSOL supports multiple, distributed animation screens which are asynchronously subscribed to one simulation model. Examples are provided for two-dimensional and three-dimensional modeling in section 5.9 on page 95. The intentional absence of a hard coded relation between a simulation object and its visualization components provides flexibility and out-of-the-box animation. This loosely coupled structure is provided by the event service (section 5.5); nearly all the other services depend on this service (see table 5.1 on page 70).

Further scientific evidence supporting the claim that this loosely coupled structure has been accomplished is presented in the two validating case studies.

- We were able to specify the optimization algorithms in the KLM case study as a service to be used by the simulation model; hence there was strong decoupling.
- We were able to link the KLM model to different external data sources, e.g. databases, proprietary KLM software, etc.
- We showed that animation panels could be distributed in both the KLM case study and in the explorative US Air Force case study.
- We were able to decouple the animation thread from the communication thread in the Dycore case study . This loosely coupled structured helped us to reach the required maximum time period of $30 \cdot 10^{-3}$ seconds (see section chapter7:section:specification:performance).

We thus conclude that we have provided scientific evidence to answer this research question positively; the event service presented in section 5.5 forms the basis for distributed, asynchronous communication which turned out to be the key factor for achieving this research objective.

Research question 4 *Can we provide scientific evidence that such simulation suite provides more effective decision support than simulation environments that are based on a 1 – 1 – 1 paradigm?*

Based on our findings we can conclude that the *usefulness* of the decision making process is improved through the current availability of several formalisms and well documented interfaces for domain specific libraries. *Usability* is improved and supported by the distributed web-based access, the separation of the modeling environment and experimentation environment and the introduction of state-of-the-art software engineering tools. *Usage* is improved and supported by the intentional absence of proprietary licenses, the absence of concealed parts in the architecture and a strong focus on the support of multiple actors. The above supports our strong conviction that the use of DSOL improves the *effectiveness* of decision making. We conclude that the suite is directly targeted at a synthetic, interdisciplinary process of problem solving.

Usage is improved and supported by the intentional absence of proprietary licenses, the absence of concealed parts in the architecture and a strong focus on the support of multiple actors.

9.2 The DSOL user community

An underlying message in this thesis has been: developing software during a Ph.D. research study imposes great risks. What will happen to DSOL tomorrow? Who

will be interested in the proprietary code developed at our university? How does one assess the quality of a one-man-made product? Considerable attention has been given to embedding DSOL in the simulation community in an attempt to mitigate these risks.

At the time of writing this thesis, the DSOL website receives 3000 visitors monthly, 50 people are subscribed to the development mailing list and the latest version has been downloaded over a 100 times. We are further pleased to state that DSOL is serving as a basis for the ongoing Ph.D. research of several colleagues in the fields of distributed gaming, main port area planning, i.e. area planning in ports and airfields, the design of real time infrastructure control strategies, e.g. train and tram infrastructure, the analysis of container terminal performance, the exploration of supply chain policies and the orchestration of distributed web services. The success of the two validating cases has rooted the use of DSOL in the cultures of TBA and KLM. Despite this we have one concern: to date, the active community developing the core services of DSOL is still relatively small (≈ 5 developers).

9.3 Recommendations for further research

In this last section of the thesis we reflect on the consequences of our N_n - N_m - N_o paradigm for the domain of systems engineering and we introduce our, partly subjective, recommendations for further research. If we consider the research conducted at our school on a more general level, two major products can be distinguished. Some researchers design an *approach* which specifies an actual process of problem solving in a particular domain. Other researchers develop (*software*) *systems* with which they aim to support an existing approach. Simulation as a method of inquiry was considered to be the foregone approach to be supported in this thesis. We nevertheless concluded that the N_n - N_m - N_o paradigm requires adjustments to be made to this approach; the sequential structure of the simulation modeling cycle presented by Banks (1998) flattens out the capriciousness of the decision making process. We therefore argue that further research should be focused in both directions, i.e. towards providing new approaches and towards designing new systems to support these approaches. A first recommendation for further research would therefore deal with the question:

Can we improve simulation as an approach for problem solving by grounding it on a multi-actor, multi-location and multi-formalism, i.e. on our N_n - N_m - N_o , paradigm?

To rephrase the above question, we state that further research should be conducted on the studio-based decision making processes. From a perspective of improving

the DSOL suite to support such an approach we extend an explicit invitation to the simulation community to use DSOL as a platform for multi-formalism modeling and as such substantially to enrich the number of supported formalisms in DSOL. Further research in this direction would deal with the question:

Can we enrich DSOL with a number of graphical user interface services for the collaborative, distributed conceptualization of specific formalisms?

We argue that further research should be focused towards providing new approaches and towards designing new systems to support these approaches.

The TBA case study (chapter 7) and the KLM case study (chapter 8) have shown the strength of an open, scalable set of simulation services. From a technical perspective it is now a small step to feed these services with organizational, e.g. transactional, data to explore operational business processes. From a decision support perspective, validation forms a serious challenge. If data from several sources and from different locations is used, new challenges concerning validation will lead to the following research question:

Can we improve the simulation approach in such a direction that the boundaries of validation are more explicit throughout the activities of experimentation and decision making?

From a perspective of infrastructure simulation, we recommend designing libraries based on the concepts used in the domain of computer gaming. Modern games feature sophisticated animation based on trackless infrastructure algorithms, e.g. binary space partition trees. This would lead to the following research question:

Can we design a library for infrastructure modeling that is based on the state-of-the-art concepts used in the domain of computer gaming?

Finally we recommend that further research is done to extend DSOL's experimentation framework to include goal functions and as such to link DSOL to external optimization services. XML-based model input and output may serve for this purpose. This would lead to the following research question:

Can we improve DSOL to accomplish interaction with state of the art optimization suites?

We finish this thesis with the more general conclusion that a service based approach to software engineering has enabled us to design and specify a full featured simulation suite, that...

We finish this thesis with the more general conclusion that a service based approach to software engineering has enabled us to design and specify a full featured simulation suite, i.e. the DSOL suite. We strongly believe that DSOL, due to its open source license, its current user community and its well documented and open project management, will survive and go on to 'outlive' this particular Ph.D. thesis trajectory.

...will survive and go on to 'outlive' this particular Ph.D. thesis trajectory.

APPENDIX

A Time advancing functions of simulators

```
200
201  /**
202   * the specification of the time advancing function of the discrete
203   * event simulator.
204   *
205   * @see nl.tudelft.simulation.dsol.simulators.Simulator#run()
206   */
207  public void run()
208  {
209      while (super.isRunning())
210      {
211          synchronized (super.semaphore)
212          {
213              SimEventInterface event = this.eventList.removeFirst();
214              super.simulatorTime = event.getAbsoluteExecutionTime();
215              super.fireEvent(SimulatorInterface.TIME_CHANGED_EVENT,
216                             super.simulatorTime, super.simulatorTime);
217              try
218              {
219                  event.execute();
220              } catch (Exception exception)
221              {
222                  Logger.severe(this, "run", exception);
223              }
224          }
225      }
226  }
```

Fig. A.1: Time advancing function of DEVSSimulator

```

60  /**
61   * @see nl.tudelft.simulation.dsol.simulators.Simulator#run()
62   */
63  public void run()
64  {
65      while (this.simulatorTime <= this.replication.getRunControl()
66              .getRunLength()
67              && isRunning())
68      {
69          synchronized (super.semaphore)
70          {
71              this.simulatorTime = this.simulatorTime + this.timeStep;
72              if (this.simulatorTime > this.replication.getRunControl()
73                  .getRunLength())
74              {
75                  this.simulatorTime = this.replication.getRunControl()
76                      .getRunLength();
77                  this.stop();
78              }
79              this.fireEvent(SimulatorInterface.TIME_CHANGED_EVENT,
80                             this.simulatorTime, this.simulatorTime);
81          }
82      }
83  }

```

Fig. A.2: Time advancing function of DESSSimulator

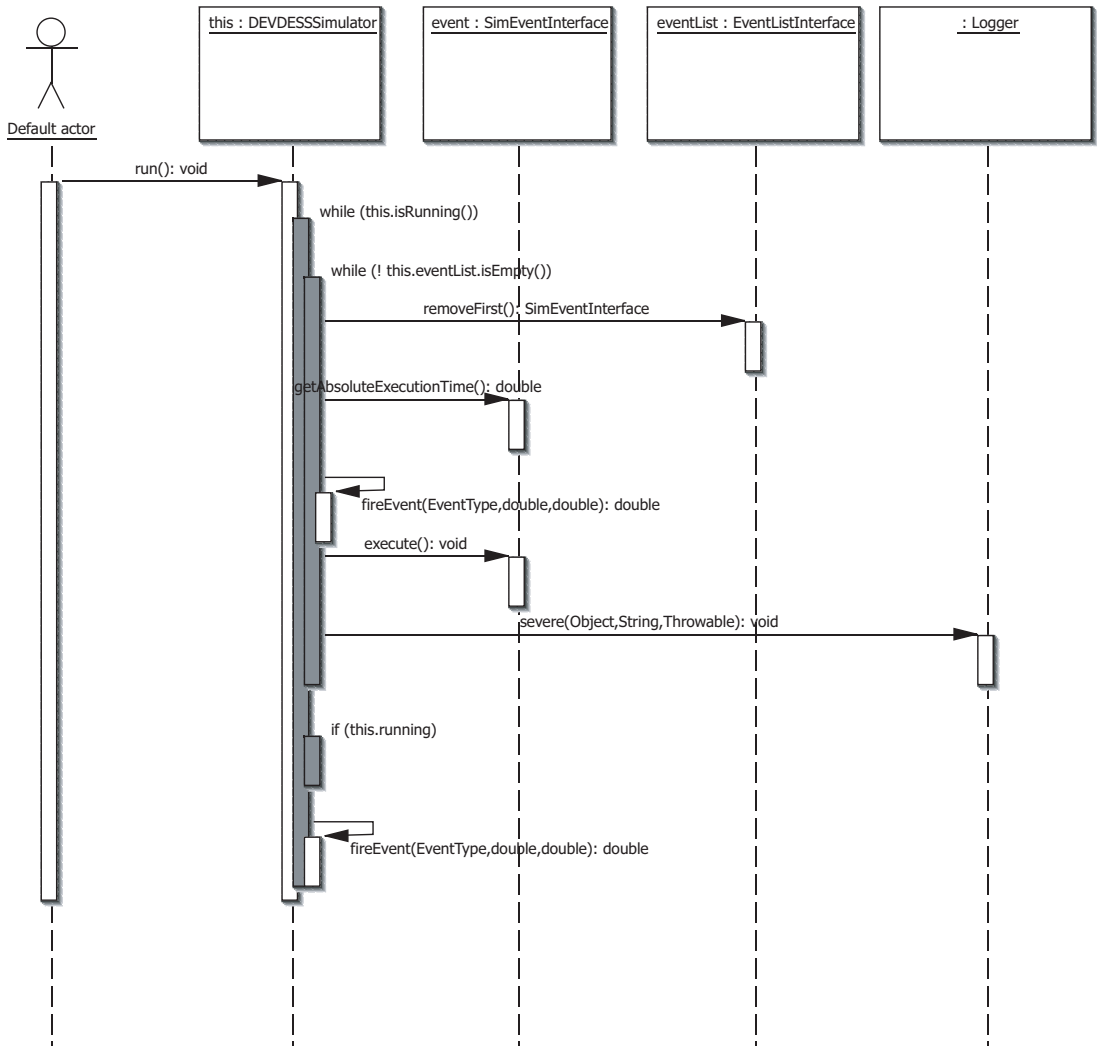


Fig. A.3: Time advancing function of DEVDESSimulator

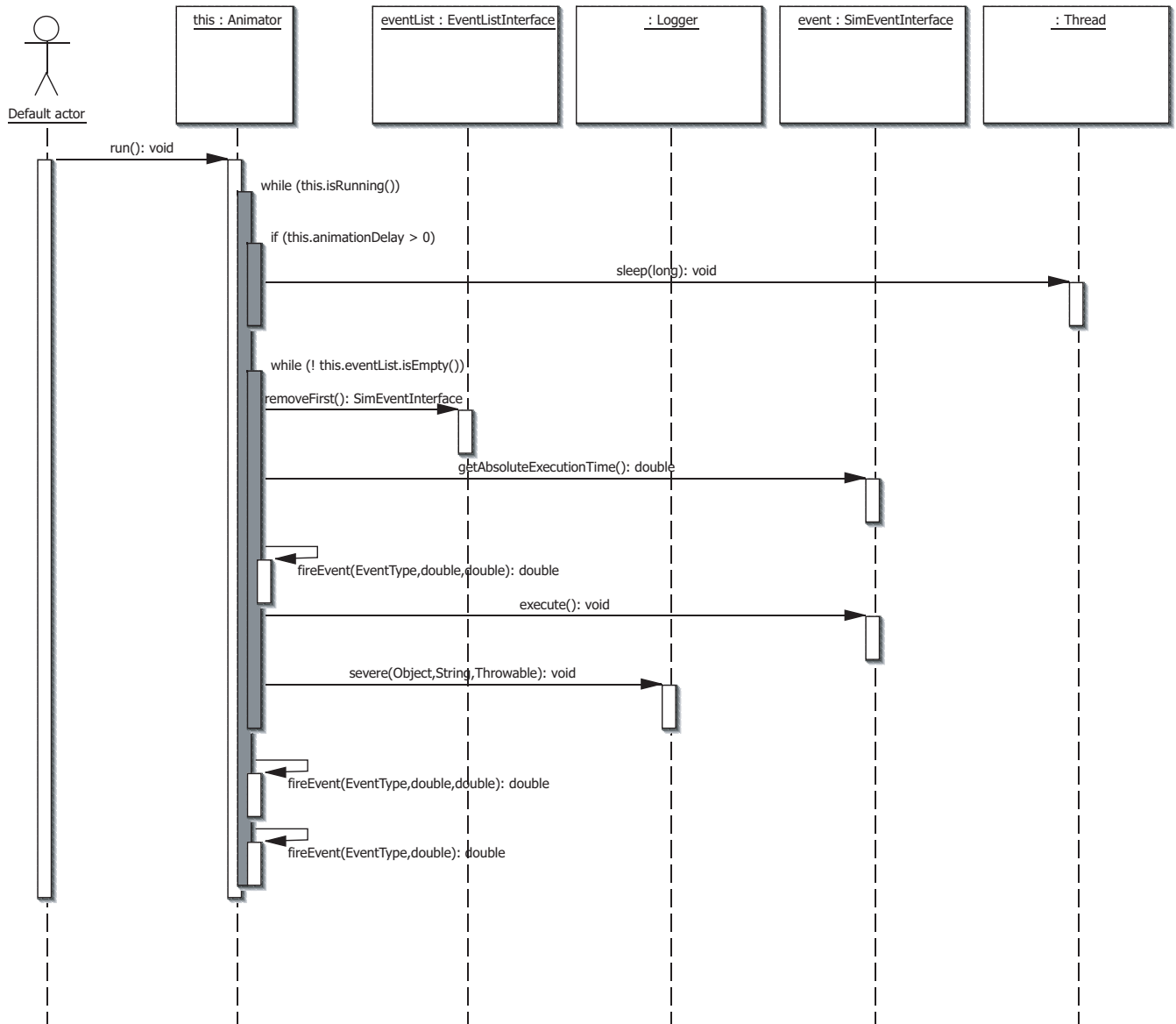


Fig. A.4: Time advancing function of Animator

B The specification of the port example

```
10 package nl.tudelft.simulation.dsol.tutorial.section45;
11
12 import nl.tudelft.simulation.dsol.formalisms.Resource;
13 import nl.tudelft.simulation.dsol.simulators.DEVSSimulatorInterface;
14
28 public class Port
29 {
34     private Resource jetties = null;
35
40     private Resource tugs = null;
41
47     public Port(final DEVSSimulatorInterface simulator)
48     {
49         super();
50         this.jetties = new Resource(simulator, "Jetties", 2.0);
51         this.tugs = new Resource(simulator, "Tugs", 3.0);
52     }
53
58     public Resource getJetties()
59     {
60         return this.jetties;
61     }
62
67     public Resource getTugs()
68     {
69         return this.tugs;
70     }
71 }

10 package nl.tudelft.simulation.dsol.tutorial.section45;
11
12 import java.rmi.RemoteException;
13 import java.util.logging.Level;
14
15 import nl.tudelft.simulation.dsol.ModelInterface;
16 import nl.tudelft.simulation.dsol.SimRuntimeException;
17 import nl.tudelft.simulation.dsol.experiment.Experiment;
18 import nl.tudelft.simulation.dsol.formalisms.devs.SimEvent;
19 import nl.tudelft.simulation.dsol.simulators.DEVSSimulator;
```

```

20 import nl.tudelft.simulation.dsol.simulators.DEVSSimulatorInterface;
21 import nl.tudelft.simulation.dsol.simulators.SimulatorInterface;
22 import nl.tudelft.simulation.language.io.URLResource;
23 import nl.tudelft.simulation.logger.Logger;
24 import nl.tudelft.simulation.xml.dsol.ExperimentParser;
38 public class BoatModel implements ModelInterface
39 {
44     public BoatModel()
45     {
46         super();
47     }
48
53     public void constructModel(final SimulatorInterface simulator)
54         throws SimRuntimeException, RemoteException
55     {
56         DEVSSimulatorInterface devsSimulator = (DEVSSimulator) simulator;
57         Port port = new Port(devsSimulator);
58
59         // We schedule boat creation
60         this.scheduleBoatArrival(0, devsSimulator, port);
61         this.scheduleBoatArrival(1, devsSimulator, port);
62         this.scheduleBoatArrival(15, devsSimulator, port);
63     }
64
74     private void scheduleBoatArrival(final double time,
75         final DEVSSimulatorInterface simulator, final Port port)
76         throws RemoteException, SimRuntimeException
77     {
78         simulator.scheduleEvent(new SimEvent(time, this, Boat.class, "<init>",
79             new Object[]{simulator, port}));
80     }
81
87     public static void main(final String[] args)
88     {
89         try
90         {
91             Logger.setLogLevel(Level.WARNING);
92             Experiment experiment = ExperimentParser
93                 .parseExperiment(URLResource.getResource("/section45.xml"));
94             experiment.setSimulator(new DEVSSimulator());

```

```
95     experiment.start();
96     } catch (Exception exception)
97     {
98         exception.printStackTrace();
99     }
100 }
101 }
```

C Java Naming and Directory Interface

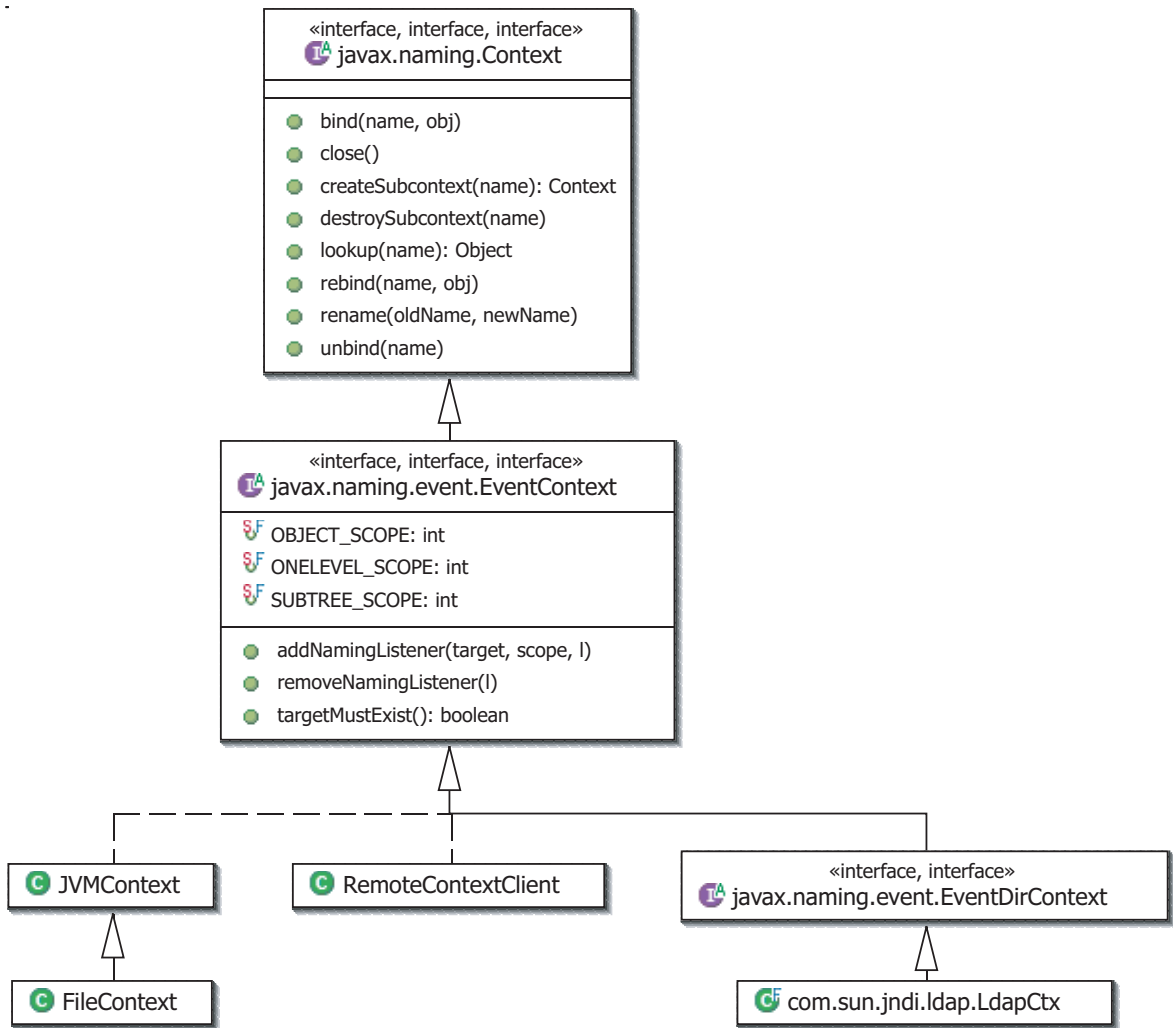


Fig. A.5: The Context interface

D DSOL experiment file

```
<?xml version="1.0" encoding="UTF-8"?>
<dsol:experimentalFrame xmlns:dsol="http://www.simulation.tudelft.nl"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <experiment>
    <model>
      <model-class>nl.tudelft.simulation.dsol.tutorial.section42.Model</model-class>
      <class-path>
        <jar-file>/tmp/tutorial.jar</jar-file>
      </class-path>
    </model>
    <simulator-class>nl.tudelft.simulation.dsol.simulators.
DEVSSimulator</simulator-class>
    <treatment>
      <startTime>2003-12-01T14:30:00</startTime>
      <timeUnit>WEEK</timeUnit>
      <warmupPeriod unit="WEEK">0</warmupPeriod>
      <runLength unit="WEEK">120</runLength>
      <properties>
        <!-- The cost properties -->
        <property key="retailer.costs.backlog" value="1"/>
        <property key="retailer.costs.holding" value="1"/>
        <property key="retailer.costs.marginal" value="3"/>
        <property key="retailer.costs.setup" value="30"/>
        <!-- The ordering policy properties -->
        <property key="policy.lowerBound" value="8"/>
        <property key="policy.upperBound" value="80"/>
      </properties>
      <replication description="replication 0">
        <stream name="default" seed="555"/>
      </replication>
      <replication description="replication 1">
        <stream name="default" seed="100"/>
      </replication>
    </treatment>
  </experiment>
</dsol:experimentalFrame>
```

E JUnit test of DSOL's discrete event list

```
1 /*
10 package nl.tudelft.simulation.dsol.eventList;
11
12 import junit.framework.Assert;
13 import junit.framework.TestCase;
14 import nl.tudelft.simulation.dsol.eventlists.EventListInterface;
15 import nl.tudelft.simulation.dsol.formalisms.devs.SimEvent;
16 import nl.tudelft.simulation.dsol.formalisms.devs.SimEventInterface;
17
18 public class EventListTest extends TestCase
19 {
20     public static final String TEST_METHOD_NAME = "test";
21
22     private EventListInterface eventList = null;
23
24     public EventListTest(final EventListInterface eventList)
25     {
26         this(EventListTest.TEST_METHOD_NAME, eventList);
27     }
28
29     public EventListTest(final String method, final EventListInterface eventList)
30     {
31         super(method);
32         this.eventList = eventList;
33     }
34
35     public void test()
36     {
37         Assert.assertNotNull(this.eventList);
38         try
39         {
40             // We fill the eventList with 500 events with random times
41             // between [0..200]
42             for (int i = 0; i < 500; i++)
43             {
44                 this.eventList.add(new SimEvent(200 * Math.random(), this,
45                     new String(), "trim", null));
46             }
47         }
48     }
49 }
```

```

78
79 // Now we assert some getters on the eventList
80 Assert.assertTrue(!this.eventList.isEmpty());
81 Assert.assertTrue(this.eventList.size() == 500);
82
83 // Let's see if the eventList was properly ordered
84 double time = 0;
85 for (int i = 0; i < 500; i++)
86 {
87     SimEventInterface simEvent = this.eventList.first();
88     this.eventList.remove(this.eventList.first());
89     double executionTime = simEvent.getAbsoluteExecutionTime();
90     Assert.assertTrue(executionTime >= 0.0);
91     Assert.assertTrue(executionTime <= 200.0);
92     Assert.assertTrue(executionTime >= time);
93     time = executionTime;
94 }
95
96 // Now we fill the eventList with a number of events with
97 // different priorities on time=0.0
98 for (int i = 1; i < 10; i++)
99 {
100     this.eventList.add(new SimEvent(0.0, (short) i, this,
101         new String(), "trim", null));
102 }
103 short priority = SimEventInterface.MAX_PRIORITY;
104
105 // Let's empty the eventList and check the priorities
106 while (!this.eventList.isEmpty())
107 {
108     SimEventInterface simEvent = this.eventList.first();
109     this.eventList.remove(this.eventList.first());
110     double executionTime = simEvent.getAbsoluteExecutionTime();
111     short eventPriority = simEvent.getPriority();
112
113     Assert.assertTrue(executionTime == 0.0);
114     Assert
115         .assertTrue(eventPriority <= SimEventInterface.MAX_PRIORITY);
116     Assert
117         .assertTrue(eventPriority >= SimEventInterface.MIN_PRIORITY);

```

```

118     Assert.assertTrue(eventPriority <= priority);
119     priority = eventPriority;
120 }
121
122 // Let's check the empty eventList
123 Assert.assertTrue(this.eventList.isEmpty());
124 Assert.assertNull(this.eventList.first());
125 Assert.assertFalse(this.eventList.remove(null));
126 Assert.assertFalse(this.eventList.remove(new SimEvent(200 * Math
127     .random(), this, new String(), "trim", null)));
128 this.eventList.clear();
129
130 // Let's cancel an event
131 this.eventList.add(new SimEvent(100, this, this, "toString", null));
132 SimEventInterface simEvent = new SimEvent(100, this, this,
133     "toString", null);
134 this.eventList.add(simEvent);
135 Assert.assertTrue(this.eventList.remove(simEvent));
136 } catch (Exception exception)
137 {
138     exception.printStackTrace();
139     Assert.fail(exception.getMessage());
140 }
141 }
142 }

```


BIBLIOGRAPHY

- Aboulafia, A. (1991). *Philosophy, Social Theory, and the Thought of George Herbert Mead*. State University of New York Press, New York: NY, USA.
- Acton, G. (2004). Great ideas in personality: metatheory. Retrieved October 21, 2004 from <http://www.personalityresearch.org/>.
- Aigner, M. and Ziegler, G. (1998). *Proofs from the book*. Springer Verlag, Berlin, Germany, 2nd edition.
- Arnold, K., Gosling, J., and Holmes, D. (2000). *The Java(TM) programming language*. Addison-Wesley, Boston: MA, USA, 3rd edition.
- Ashby, W. (1956). *An introduction to cybernetics*. John Wiley & Sons, New York: NY, USA.
- Balci, O. (1988). The implementation of four conceptual frameworks for simulation modeling in high-level languages. In Abrams, M., Haigh, P., and Comfort, J., editors, *Proceedings of the 20th conference on Winter simulation*, pages 287–295, San Diego: CA, USA. IEEE, ACM Press.
- Balci, O. (1995). Principles and techniques of simulation validation, verification, and testing. In Lilegdon, W., Goldsman, D., Alexopoulos, C., and Kang, K., editors, *Proceedings of the 27th conference on Winter simulation*, pages 147–154. IEEE, ACM Press.
- Banks, J. (1998). Principles of simulation. In Banks, J., editor, *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, pages 3–31. Wiley-Interscience, New York: NY, USA.
- Barr, J. (2003). Business intelligence 2003: Are your bi systems making you smarter? Retrieved October 21, 2004 from http://www.cioinsight.com/print_article/0,1406,a=42221,00.asp.
- Birtwistle, G. M. (1979). *Discrete Event Modelling on Simula*. Macmillan Press, Houndmills, UK.

- Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The unified modeling language user guide*. Addison-Wesley, Indianapolis: IN, USA.
- Bosman, A. (1977). *Een metatheorie over het gedrag van organisaties*. Stenfert Kroese, Leiden, the Netherlands. (in Dutch).
- Boyson, S., Corsi, T., and Verbraeck, A. (2003). The e-supply chain portal: a core business model. *Transportation Research Part E*, 39:175–192.
- Breitenecker, F. (1990–2004). Simulation news europe: comparisons. Retrieved October 21, 2004 from <http://www.argesim.org/comparisons/index.html>.
- Briggs, R., de Vreede, G.-J., and Nunamaker, J. (2003). Collaboration engineering with thinklets to pursue sustained success with group support systems. *Journal of Management Information Systems*, 19:31 – 64.
- Brussaard, B. and Tas, P. (1980). Information and organization policies in public administration. In Lavington, S., editor, *Proceedings of IFIP Congress 80*, pages 821–826, Tokyo, Japan. International Federation for Information Processing, North-Holland.
- Burn, O. (2001-2003). Checkstyle overview. Retrieved October 21, 2004 from <http://checkstyle.sourceforge.net/>.
- Buxton, J. and Laski, J. (1962). Control and simulation language. *The computer journal*, 5:194–199.
- Cai, W., Lee, F., and Chen, L. (1999). An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 82–89. IEEE Computer Society.
- Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *Computing surveys*, 17(4):471–522.
- Churchman, C. (1971). *The design of Inquiring Systems: Basic Concepts of Systems and Organizations*. Basic Books, New York: NY, USA.
- Corré, A. (1992). Lecture on the lingua franca. *Contacts between Cultures: West Asia and North Africa*, 1(1):140–145.
- Cournot, A. (1838). *Recherches sur les principes mathématiques de la thrie des richesses*. Hachette, Paris, France.
- Dahl, O.-J. (2002). The birth of object orientation: the simula languages. In Broy, M. and Denert, E., editors, *Software Pioneers*, pages 78–91. Springer, Berlin, Germany.

- Daum, B. and Horak, C. (1999). *The XML shockwave*. Software AG, Darmstadt, Germany.
- Delone, W. and McLean, E. (1992). Information system success: the quest for the dependent variable. *Information Systems Research*, 3(1):60–95.
- Delone, W. and McLean, E. (2003). The delone and mclean model of information systems success: A ten-year update. *Journal of Management Information Systems*, 19(4):9–30.
- Deng, L. and Xu, H. (2003). A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Transactions on modeling and computer simulation*, 13(4):299–309.
- Denning, P. (1997). A new social contract for research. *Communications of the ACM*, 40(2):132–134.
- Dick, B. (1999). What is action research. Retrieved October 21, 2004 from <http://www.scu.edu.au/schools/gcm/ar/whatisar.html>.
- Dycore (2004). Dycore - productinfo bekistingsplaatvloer, kanaalplaatvloer en ribbenvloer. Retrieved October 21, 2004 from <http://www.dycore.nl/>. (in Dutch).
- Eccleston, J. (2002). The net-centric supply chain. Retrieved October 22, 2004 from http://www.afei.org/brochure/2AF2/james_eccleston.pdf.
- Eckel, B. (2000). *Thinking in C++*. Prentice-Hall, New York: NY, USA, 2nd edition.
- Eckel, B. (2004a). `< T extends NonGenericType>` and `Class < T >`. Retrieved February 28, 2005 from <http://www.mindview.net/WebLog/log-0062>.
- Eckel, B. (2004b). Why java needs latent typing. Retrieved February 28, 2005 from <http://www.mindview.net/WebLog/log-0063>.
- Eisenhardt, K. (1989). Building theories from case study research. *Academy of management review*, 14(4):532–550.
- Faires, J., Burden, R., Faires, D., Pirtle, B., and Sandberg, K. (2002). *Numerical Methods*. Brooks Cole, Boston: MA, USA, 3rd edition.
- Fischman, G. (1973). *Concepts and methods in discrete event digital simulation*. John Wiley & Sons, New York: NY, USA.

- Frog (2004). Park-a-car. Retrieved October 22, 2004 from <http://www.frog.nl/cargo.php?f=parkacar\&s=capplications>.
- Fujimoto, R. (2000). *Parallel and distributed simulation systems*. John Wiley & Sons, Cambridge: MA, USA.
- Galliers, R. (1992). Choosing information systems research approaches. In Galliers, R., editor, *Information systems research: issues, methods, and practical guidelines*, pages 144–162. Blackwell Scientific Publications, Oxford, UK.
- Gove, P., editor (2002). *Webster's third new international dictionary, unabridged*. Merriam-Webster, 3rd edition.
- Healy, K. and Kilgore, R. (1997). Silk: a Java-based process simulation language. In *Proceedings of the 29th conference on Winter simulation*, pages 475–482, Atlanta: GA, USA. ACM Press.
- Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1).
- Hintikka, J. (1975). Rudolf Carnap, logical empiricist. In *Materials and Perspectives*. D. Reidel Publishing Company, Dordrecht, the Netherlands.
- Hlupic, V. (1993). *Simulation modeling software approaches to manufacturing problems*. PhD thesis, London School of Economics, London, UK.
- Hoare, C. A. R. (1968). Record handling. In Genuys, F., editor, *Programming Languages: NATO Advanced Study Institute*, pages 291–347. Academic Press, London, UK.
- Holbaek-Hansen, E. (1975). *System description and the Delta language*. NCC, Oslo, Norway.
- Hoschek, W. (2002). The colt distribution. Retrieved October 21, 2004 from <http://hoschek.home.cern.ch/hoschek/colt/>.
- Jacobs, J. and Jacobs, P. (2004). Gisbeans: a Java library for geographical information systems. Retrieved October 21, 2004 from <http://gisbeans.sourceforge.net>.
- Jacobs, P. (2004). Specifying the sne comparisons in dsol. Retrieved October 25, 2004 from <http://www.simulation.tudelft.nl/dsol/sne>.

- Jacobs, P., Lang, N., and Verbraeck, A. (2002). A distributed Java based discrete event simulation architecture. In Yucesan, E., Chen, C.-H., Snowdon, J., and Charnes, J., editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 793–800, San Diego: CA, USA. IEEE, ACM Press. Retrieved October 21, 2004 from <http://www.informs-cs.org/wsc02papers/102.pdf>.
- Jacobs, P. and Verbraeck, A. (2004a). *Mastering DSOL: a Java based suite for simulation*. Delft University of Technology, Delft, the Netherlands, 2nd edition. Retrieved October 21, 2004 from <http://www.simulation.tudelft.nl/dsol>.
- Jacobs, P. and Verbraeck, A. (2004b). Single-threaded specification of process-interaction formalism in Java. In Ingalls, R. G., Rossetti, M. D., Smith, J. S., and Peters, B. A., editors, *Proceedings of the 2004 Winter Simulation Conference*, Washington: DC, USA. IEEE, ACP Press.
- Jacobs, P., Verbraeck, A., and Mulder, J. (2005a). Flight scheduling at klm. In Kuhl, M. E., Steiger, N. M., Armstrong, F. B., and Joines, J. A., editors, *Proceedings of the 2005 Winter Simulation Conference*, Orlando: FL, USA. IEEE, ACM Press.
- Jacobs, P., Verbraeck, A., and Rengelink, W. (2005b). Emulation with dsol. In Kuhl, M. E., Steiger, N. M., Armstrong, F. B., and Joines, J. A., editors, *Proceedings of the 2005 Winter Simulation Conference*, Orlando: FL, USA. IEEE, ACM Press.
- James, W. (1890). *Principles of Psychology*, volume 1–2. unknown. Retrieved October 21, 2004 from <http://psychclassics.yorku.ca/James/Principles/index.htm>.
- Joy, B., Steele, G., Gosling, J., and Bracha, G. (2000). *Java(TM) language specification*. Addison-Wesley, Boston: MA, USA, second edition.
- Kaufmann, R. and Janzen, D. (2003). Implications of test-driven development: a pilot study. In Crocker, R. and G. L. Steele, J., editors, *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299, Anaheim: CA, USA. ACM, ACM Press.
- Kazi, I. H., Jose, D. P., Ben-Hamida, B., Hescott, C. J., Kwok, C., Konstan, J. A., Lilja, D. J., and Yew, P.-C. (2000). Javiz: A client/server Java profiling tool. *IBM Systems Journal*, 39(1).
- Keen, P. and Sol, H. (2005). *Rehearsing the future*. draft version.

- Kiviat, P. (1967). Digital computer simulation: modeling concepts. In *RAND Memo RM-5378-PR*, Santa Monica: CA, USA. RAND Corporation.
- Klir, G. (1985). *Architecture of systems problem solving*. Plenum Press, New York: NY, USA.
- KLM (2004). KLM, annual report 2003/2004. Retrieved October 21, 2004 from http://www.klm.com/corporate_en.
- Knuth, D. (1998). *The art of computer programming*, volume 2. Addison-Wesley, Boston: MA, USA, 3rd edition.
- Koningsveld, H. (1987). *Het verschijnsel wetenschap*. Boom, Amsterdam, the Netherlands, 8th edition. (in Dutch).
- Kuljis, J. and Paul, R. J. (2000). A review of web based simulation: whither we wander? In *Proceedings of the 32nd conference on Winter simulation*, pages 1872–1881, Orlando: FL, USA. SCS.
- Law, A. and Kelton, W. (2000). *Simulation modeling and analysis*. Mc Graw Hill, Singapore, Singapore, third edition.
- L'Ecuyer, P. (1997). Uniform random number generators: a review. In Andradóttir, S., Healy, K., Withers, D. H., and Nelson, B., editors, *Proceedings of the 29th conference on Winter simulation*, pages 127–134, Atlanta: GA, USA. ACM Press.
- L'Ecuyer, P., Blouin, F., and Couture, R. (1993). A search for good multiple recursive random number generators. *ACM Transactions on modeling and computer simulation*, 3(2):87–98.
- L'Ecuyer, P., Meliani, L., and Vaucher, J. (2002). Ssj: A framework for stochastic simulation in Java. In Yucesan, E., Chen, C.-H., Snowdon, J., and Charnes, J., editors, *Proceedings of the 2002 Winter Simulation Conference*, San Diego: CA, USA. IEEE, ACM Press. Retrieved October 24, 2004 from <http://www.informs-cs.org/wsc02papers/030.pdf>.
- L'Ecuyer, P. and Simard, R. (1999). Beware of linear congruential generators with multipliers of the form $a = \pm 2q \pm 2r$. *ACM Transactions on Mathematical Software*, 25(3):367–374.
- Lee, A. (2000). Systems thinking, design science, and paradigms: Heeding three lessons from the past to resolve three dilemmas in the present to direct a trajectory for future research in the information systems field. In *Proceedings of the*

11th International Conference on Information Management. IEEE Computer Society.

- Lee, R. and Tepfenhart, W. (2002). *Practical object-oriented development with UML and Java*. Prentice-Hall, Upper Saddle River: HJ, USA.
- Lehmer, D. (1951). Mathematical methods in large-scale computing units. In *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*, pages 141–146, London, UK. Harvard University Press.
- Lieberman, H., Ungar, D., and Stein, L. (1988). The treaty of orlando: a shared view of sharing. In Kim, W. and Lochovsky, F., editors, *Object-Oriented Concepts, Applications and Databases*. Addison-Wesley, Boston: MA, USA.
- Lindholm, T. and Yellin, F. (1999). *The Java(TM) virtual machine specification*. Addison-Wesley, London, UK, 2nd edition.
- Link, J. and Frolich, P. (2003). *Unit testing in Java*. Morgan Kaufmann, San Francisco: CA, USA.
- Mandojana, J., Herman, K., and Zulinski, R. (1990). A discrete/continuous time-domain analysis of a generalized class e amplifier. *IEEE Transactions on Circuits and Systems*, 37(8):1057–1060.
- March, S. and Smith, G. (1995). Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266.
- Matic, N. (2001). *Introduction to PLC controllers*. mikroElektronika, Belgrade, Serbia. Retrieved October 28, 2004 from <http://www.mikroelektronika.co.yu/english/product/books/PLCbook/plcbook.htm>.
- Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on modeling and computer simulation*, 8(1):3–30.
- Meyer, B. (1992). Applying "design by contract". *IEEE Computer*, 25(10):40–51.
- Meyer, B. (1997). *Object-oriented software construction*. Prentice-Hall, New York: NY, USA, 2nd edition.
- Mitroff, I., Betz, F., Pondy, L., and Sagasti, F. (1974). On managing science in the system age: two schemes for the study of science as a whole systems phenomenon. *TIMS interfaces*, 4(3):46–58. Reprinted in *Systems and Management Annual*, C. W. Churchman (ed.), 1975.

- Modicon (1996). Modbus protocol reference guide. Technical Report PI-MBUS-300, Modbus-IDA. Rev. J.
- Morris, D. (2003). Junit automates Java testing. *IT Jungle*, 2(22). Retrieved October 28, 2004 from <http://www.itjungle.com/mpo/mpo110603-story01.html>.
- Nance, R. E. (1981). The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179. Special issue on simulation modeling and statistical computing.
- Nance, R. E. (1995). Simulation programming languages: An abridged history. In Alexopoulos, C., Kang, K., Lilegdon, W., and Goldsman, D., editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 1307–1313, Arlington: VA, USA. IEEE.
- Newell, A. and Simon, H. (1963). Gps, a program that simulates human thought. In Feigenbaum, E. and Feldman, J., editors, *Computers and Thought*, pages 279 – 293. Mc Graw Hill.
- Nutt, P. (2002). *Why Decisions Fail*. Berrett-Koehler Publishers, San Francisco: CA, USA.
- Ören, T. and Zeigler, B. (1979). Concepts for advanced simulation methodologies. *Simulation*, 32(3):69–82.
- Overstreet, C. and Nance, R. (1986). World view based discrete event model simplification. In Elzas, M. S., Oren, T. I., and Zeigler, B. P., editors, *Modelling and Simulation Methodology in the Artificial Intelligence Era*, pages 165–179, Amsterdam, the Netherlands. North-Holland.
- Page, E., Moose, R., and Griffin, S. (1997). Web based simulation in simjava using remote method invocation. In A. Andrattotir, K.J. Healy, D. W. and Nelson, B., editors, *Proceedings of the 1997 Winter Simulation Conference*, Atlanta: GA, USA. IEEE, ACM Press.
- Papazoglou, M. and Dubray, J.-J. (2004). A survey of web service technologies. Dit-04-058, Informatica e Telecomunicazioni, University of Trento, Trento, Italy.
- Papazoglou, M. and Geogakopoulos, D. (2003). Service oriented computing. *Communications of the ACM*, 46(10):25–28.
- Roache, P. (1998). *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque: NM, USA.

- Samson, D., Huibers, L., and Jacobs, P. (2004). Flexible assembly system - DSOL specification. <http://www.simulation.tudelft.nl/dsol/sne/publications/c2.pdf>.
- SAP (2004). management-cockit. Retrieved October 21, 2004 from <http://help.sap.com/>.
- Schiess, C. (2001). Emulation: debug it in the lab — not on the floor. In Rohrer, M., Medeiros, D., and Grabau, M., editors, *Proceedings of the 33rd conference on Winter simulation*, pages 1463–1465, Arlington: VA, USA. IEEE, ACM Press.
- Schludermann, H., Kirchmair, T., and Vorderwinkler, M. (2000). Soft-commissioning: hardware-in-the-loop-based verification of controller software. In Fishwick, P., Kang, K., Joines, J., and Barton, R., editors, *Proceedings of the 32nd conference on Winter simulation*, pages 893–899, Orlando: FL, USA. IEEE, ACM Press.
- Seddon, P. (1997). A respecification and extension of the delone and mclean model of is success. *Information Systems Research*, 8(3):240–253.
- Shannon, R. (1975). *Systems simulation: the art and science*. Prentice-Hall, Indianapolis: IN, USA.
- Simon, H. (1955). A behavioral model of rational choice. *Quarterly journal of economics*, 69:99–118.
- Simon, H. (1976). From substansive to procedural rationality. In Latsis, S., editor, *Method and appraisal in economics*, pages 129–149. Cambridge University Press, Cambridge, UK.
- Simon, H. (1977). *Models of discovery*, volume LIV. D. Reidel publishing company, Dordrecht, the Netherlands.
- Simon, H. (1996). *The Sciences of the Artificial*. MIT Press, Cambridge: MA, USA, 3rd edition.
- Sing, L. (2000). *Professional Jini*. Wrox Press Ltd., Birmingham, UK.
- Sokal, N. and Sokal, A. (1975). Class e - a new class of high-efficiency tuned single-ended switching power amplifiers. *IEEE Journal of Solid-State Circuits*, SC-10(3):168–176.
- Sol, H. (1982). *Simulation in information systems development*. PhD thesis, Rijksuniversiteit Groningen, Groningen, the Netherlands.

- Sun Microsystems (2001a). Java logging apis. Retrieved May 11, 2005 from <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/>.
- Sun Microsystems (2001b). Java naming & directory interface apis. Retrieved May 25, 2005 from <http://java.sun.com/products/jndi/docs.html#12>.
- Tarr, P. and Ossher, H. (2001). Workshop on advanced separation of concerns in software engineering. In Müller, H. A., editor, *Proceedings of the 23rd International Conference on Software Engineering*, pages 778–779, Toronto, Canada. IEEE Computer Society.
- TBA Nederland (2000). Tba nederland specialised in simulation of factories, harbours, airports and railsystems. Retrieved October 21, 2004 from <http://www.tbanederland.nl/default.asp>.
- Tewoldeberhan, T., Verbraeck, A., Valentin, E., and Bardonnnet, G. (2002). An evaluation and selection methodology for discrete-event simulation software. In Yucesan, E., Chen, C.-H., Snowdon, J., and Charnes, J., editors, *Proceedings of the 2002 Winter Simulation Conference*, San Diego: CA, USA. IEEE, ACM Press. Retrieved October 21, 2004 from <http://www.informs-cs.org/wsc02papers/010.pdf>.
- Tsichritzis, D. (1997). The dynamics of innovation. *Beyond calculation: the next fifty years*, pages 259–265.
- van Duin, E. (2005). Making Opium more addictive. M.sc. thesis, Delft University of Technology, Delft, the Netherlands. draft version.
- van Wendel de Joode, R. (2005). *Understanding open source communities*. PhD thesis, Technische Universiteit Delft, Delft, the Netherlands.
- Vangheluwe, H. and de Lara, J. (2002). Meta-models are models too. In Yucesan, E., Chen, C.-H., Snowdon, J., and Charnes, J., editors, *Proceedings of the 2002 Winter Simulation Conference*, San Diego: CA, USA. IEEE, ACM Press. Retrieved October 21, 2004 from <http://www.informs-cs.org/wsc02papers/076.pdf>.
- Verbraeck, A. (2004). Real-time visualization and modeling of supply chains. In Boyson, S., T.Corsi, and Harrington, L., editors, *in real-time: managing the new supply chain*, chapter 7. Praeger Publishers, Westport: CT, USA.
- Walls, J., Widmeyer, G., and Sawy, O. E. (1992). Building an information system design theory for vigilant eis. *Information Systems Research*, 3(1):36–59.

- Weisstein, E. (1999). Ode solving. Retrieved November 24, 2004 from MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/topics/ODESolving.html>.
- Whorter, S., Baker, B., and Malan, G. (1997). Simulation system for control software validation. In *Proceedings of the 1997 SCS Simulation Multiconference*, Atlanta: GA, USA.
- Wilson, R. and Keil, F. (1999). *The MIT encyclopedia of the cognitive sciences*. Bradford Books, Cambridge: MA, USA.
- Wirth, N. (1979). *Algorithmen und datenstrukturen*. Teubner Studienbücher, Stuttgart, Germany.
- Wood, D. (1992). *Data structures, algorithms and performance*. Addison-Wesley, Boston: MA, USA.
- Zeigler, B. (1976). *Theory of modeling and simulation*. Academic Press, San Diego: CA, USA.
- Zeigler, B. (1984). *Multifaceted modeling and discrete event simulation*. Academic Press, London, UK.
- Zeigler, B., Praehofer, H., and Kim, T. (2000). *Theory of modeling and simulation*. Academic Press, San Diego: CA, USA, 2nd edition edition.
- Zeigler, B. and Sarjoughian, H. (2005). *Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models*. Arizona Center for Integrative Modeling and Simulation, draft version 3 edition. Retrieved March 22, 2005 from http://www.acims.arizona.edu/SOFTWARE/devsjava_licensed/CBMSManuscript.zip.

Author index

- A**
- Aboulafia, M. 10
Acton, G.S. 9
Aigner, M. 12
Arnold, K. 71, 89
Ashby, R. 35
- B**
- Balci, O. 53, 54, 78, 99
Banks, J. 56, 138
Bardonnet, G. 5, 18
Barr, J. 1
Ben-Hamida, B. 128
Betz, F. 4
Birtwistle, G.M. 29, 54, 85
Blouin, F. 89
Booch, G. 37, 39, 44, 70
Bosman, A. 36
Boyson, S. 16–18, 23
Bracha, G. 71, 127
Brussaard, B.K. 36
Burden, R.L. 87
Burn, O. 126
Buxton, J.N. 54
- C**
- Cai, W. 23
Cardelli, L. 41
Carnap, R. 9
Chen, L. 23
Churchman, C.W. 4, 36
Corré, A.D. 68
Corsi, T. 16–18, 23
Cournot, A.A. 2
Couture, R. 89
- D**
- Dahl, O.-J. 38, 39
de Lara, J. 13, 51, 58, 59
- DeLone, W.H. 10
Deng, L.Y. 89
Denning, P.J. 10
Descartes, R. 9
Dick, B. 12
Dubray, J.-J. 7, 57
Duhem, P.M.M. 10
- E**
- Ecclesia, J.T. 15, 18
Eckel, B. 39, 42, 43, 65, 74
Ecuyer, P. *see* L'Ecuyer, P.
Eisenhardt, K.M. 12
El Sawv, O. 11
- F**
- Faires, J.D. 87
Fishman, G.S. 49
Frolich, P. 128
- G**
- Galliers, R.D. 11
Georgakopoulos, D. 7, 8
Gosling, J. 71, 89, 127
- H**
- Healy, K.J. 66
Hegel, G.F. 4
Herman, K.J. 111
Hescott, C.J. 128
Hevner, A.R. 10, 11
Hintikka, J. 9
Hlupic, V. 5, 18, 49
Hoare, C.A.R. 38
Holbaek-Hansen, E. 35, 37, 48
Holmes, D. 71, 89
Hoschek, W. 114
Huibers, L. 108
Hume, D. 9

J
 Jacobs, J.P.M. 149, 164
 Jacobs, P.H.M. 23, 66, 80, 82, 84, 98,
 101, 108, 134, 149, 153, 164
 Jacobson, I. 37, 39, 44, 70
 James, W. 2
 Janzen, D. 126
 Jose, D.P. 128
 Joy, B. 71, 127

K
 Kant, I. 10
 Kaufmann, R. 126
 Kazi, I.H. 128
 Keen, P.W.G. 1, 5, 6, 36, 47, 60
 Keil, F. 9
 Kelton, W.D. 55, 91, 94
 Kilgore, R.A. 66
 Kim, T. 13, 48, 51, 53, 58, 59, 74
 Kirchmair, T. 136, 138
 Kiviat, P.J. 49
 Klir, G.J. 9, 35–37, 48, 50
 Knuth, D.E. 88, 89
 Koningsveld, H. 10
 Konstan, J.A. 128
 Kowk, C. 128
 Kuljis, J. 65

L
 L'Ecuyer, P. 65, 89
 Lakatos, I. 10
 Lang, N.A. 23, 80
 Laski, J.G. 54
 Law, A.M. 55, 91, 94
 Lee, F.B.S. 10, 23
 Lee, R.C. 39
 Lehmer, D.H. 89
 Leibnitz, G.W. 4
 Lieberman, H. 43
 Lilja, D.J. 128
 Lindholm, T. 82, 83

Link, J. 128
 Lock, J. 4

M
 Mandojana, J.C. 111
 March, S.T. 10, 11
 Matsumoto, M. 89
 McLean, E. 10
 Meyer, B. 38, 44
 Mitroff, I.I. 4, 99
 Morris, D. 128
 Mulder, J.B.P. 153

N
 Nance, R.E. 13, 49, 54, 59
 Newell, A. 9
 Nishimura, T. 89
 Nutt, P.C. 4
 Nygaard, K. 38

O
 Ören, T.I. 55, 56
 Ossher, H. 38
 Overstreet, C.M. 54

P
 Papazoglou, M.P. 7, 8, 57
 Park, J. 10, 11
 Paul, R.J. 65
 Pirtle, B. 87
 Pondy, L. 4
 Popper, C. 10
 Praehofer, H. 13, 48, 51, 53, 58, 59, 74

R
 Ram, S. 10, 11
 Rengeling, W. 134
 Roache, P.J. 56, 99
 Rumbaugh J. 37, 39, 44, 70

S
 Sagasti, F. 4
 Samson, D. 108

Sandberg, K.	87
Sarjoughian, H.S.	65
Schiess, C.	136
Schludermann, H.	136, 138
Seddon, P.B.	10
Shannon, R.E.	4, 12, 47, 57
Simard, R.	89
Simon, H.A.	2, 9, 10, 35–37
Sing, L.	44
Singer, E.A.	4
Smith, G.	10, 11
Sokal, A.D.	111
Sokal, N.O.	111
Sol, H.G.	1, 4–6, 36, 37, 47, 55–57, 59, 60, 99
Steele, G.	71, 127
Stein, L.	43

T

Tarr, P.	38
Tas, P.A.	36
Tepfenhart, W.M.	39
Tewoldeberhan, T.	5, 18
Tsichritzis, D.	10

U

Unger, D.	43
----------------	----

V

Valentin, E.	5, 18
van Duin, E.	161
Vangheluwe, H.	13, 51, 58, 59
Verbraeck, A.	5, 16–18, 23, 24, 66, 80, 82, 84, 98, 134, 153
Vorderwinkler, M.	136, 138

W

Walls, J.	11
Wegner, P.	41
Weisstein, E.W.	88, 89
Widmeyer, G.	11
Wilson, R.A.	9

Wirth, N.	81
Wood, D.	81

X

Xu, H.	89
-------------	----

Y

Yellin, F.	82, 83
Yew, P.C.	128

Z

Zeigler, B.P.	12, 13, 48, 50, 51, 53, 55, 58, 59, 65, 74
Zulinski, R.E.	111

Subject index

A

abstract class 74
acceptance *see* satisfactory level
accessibility 43
activism 9
activity scanning 54
actor 48
Adams' method 89, 104
administrative sciences 2
aggregation 39
Air France 153
Algol 60 38
Animator 78, 180
Arena 61, 154
asynchronous communication . . . 44, 70
Automod 61

B

backdrop 50
bad child 43
behavior 49
behavioral-science 10
Bernoulli distribution 91
Beta distribution 91
Binomial distribution 91
body of knowledge 35, 36
bureaucratic principle 43
business process 1

C

cancel measure 158
CERN 114
Checkstyle 126
class 38, 40
classical economic theory 2
closed under coupling 53
Coad-Yourdon 38
coercion 41

COLT 114
combined device 140
composite service 8
conceptual model 3
conceptualization 56
conceptualization freedom 58
conservative activism 10
Constant distribution 91
constantpool 82
construct 11
control state 81
control system 133, 138
control theory 35
controlled system 138
conventionalism
 see activism 9
cybernetics 35

D

database tier 19
dead reckoning 23
decision process 1
delegation 43
design 11
design by contract 44
design-science 10
DESS 51, 87–88
DESSSimulator 78
DEVDESSimulator 78
DEVDESSSimulator 179
DEVS 51, 78–81
DEVSSimulator 78
Discrete Constant distribution . . . 91
distributed computing 19
domain of inquiry 35
DSOL 63–98
dsol-gui service 68
dsol-xml service 68

DTSS 51
 DX-120 generator 89
 Dycore 133
 dynamic class downloading 44

E

Eiffel 45
 eM-Plant 61, 133
 Empirical continuous distribution . 91
 Empirical discrete distribution 91
 empirical model 4
 emulation 133
 encapsulation 40
 epistemological level 48
 Erlang distribution 91
 Euler's method 89, 104
 event 50
 event listener 70
 event producer 70
 event project 70–74
 event scheduling 53, 54
 experiment 55, 56
 experimental frame 48, 55
 experimentation 57
 expert validation 12, 57
 exploration 11
 Exponential distribution 91
 external validity 12

F

falsificationism 9
 formalism 50
 framestack 84
 framework for M & S 49
 Frog navigation 26–33
 Fusion 38

G

Gamma distribution 91
 general problem solver 9
 general systems research 35
 generalization 12

Geometric distribution 91
 Gill's method 89, 104
 Gisbeans 149
 good child 43

H

hard coded 24
 Heun's method 89, 104
 horizontally layered ... *see* subsystem

I

inclusion 41, 42
 inductivists 9
 information hiding 40
 information theory 35
 inheritance 42
 input device 140
 inquiring system 4
 inquiry system 4
 instant 50
 instantiation 11
 interface 45
 interpretivism
 see activism 9
 interval 50

J

Java interpreter 82
 JNDI 69
 JUnit testing 128

K

Kantian 10
 KLM 153–167

L

laboratory 37
 late binding 44
 linear congruential method 89
 locality 53
 LogNormal distribution 91

M

maintenance measure 158
mathematical systems theory 35
Mersenne Twister 89
metatheoretical freedom 58
method 11
methodology 35
Milne's method 89, 104
mock-up device 136
Modbus 140
model 11
model cycle 4
modeling construct 50
modeling freedom 58
modifier 40
modularity 37
multi-channeling 19
multi-formalism 51
municipality of Rotterdam 26

N

Negative Binomial distribution 91
neutrality 12
non-stakeholders 47
Normal distribution 91

O

object 38
object-orientation 38–39
observation *see* backdrop
ODE 87
OMT 38
OOSE 38
operational plan 154
OPiuM 153–157
output device 140
overloading 41

P

parametric polymorphism 41
Park-A-Car 26
passivism 9

Pearson 5 distribution 91
Pearson 6 distribution 91
perceived problem 3
PLC 134
Poisson distribution 91
portal tier 18
probabilists 9
problem solving 35
process 50
process interaction 54, 81–86
Profiling 128
pseudo random number generators 88–
91

R

rationality
 procedural 2
 substantive 2
real system 36
RealtimeClock 78
Red-Black binary tree 81
reflection 45, 80
register 139
replication 55, 56
replicative validation 57
research approach 9–12
research instruments 9, 11
research philosophy 9
research strategy 9, 10
reserved measure 158
revolutionary conventionalism 10
Royal Dutch Airlines *see* KLM
RPC 19
run control 55, 56
Runge Kutta - Cash Carp method 89,
104
Runge Kutta - Fehlberg method .. 89,
104
Runge Kutta 3 method 89, 104
Runge Kutta 4 method 89, 104
runtime 44

S

satisficing level 2
scheduled method invocation 80
schools of thought 9
scientific method 9
security 18
sensory impressions 9
sensory observations 10
serializable 71
service 7, 57
service oriented architecture 7
service oriented computing 7
service oriented society 7
service providers 7
sheet piling floor 134
side loader 26
SimEvent 71, 80
Simula 38
Simula 1 *see* Simula
Simula 67 *see* Simula
simulation 47, 48
simulation model 48
simulation time *see* system time
simulator 48, 74–78
SNE 99–131
solution finding freedom 58
source system 48
span 50
specification 56
stack swapping 82
stakeholders 47
state 49, 50
state transition 49
state-time relations 13
statistical distributions 91
structural validation 57
structure
 dynamic 49
 static 49
subscription 44, 70

subsystem 37
subtype polymorphism . *see* inclusion
swap measure 158
system 35
system time 50
systems
 engineering 35–37
 sciences 35
systems design 35

T

TBA Nederland 133
theorem 12
theory-laden 9
timed event 71
treatment 55
Triangular distribution 91

U

UML 39
Uniform distribution 91
USAF 15–24

V

validation 56
verification 56
vertically partitioned . *see* subsystem
visibility 43

W

weak reference 71
web service 8
Weibull distribution 91
world view 50

SUMMARY

In this thesis we argue that a new paradigm is needed for the design of decision support systems to support unstructured decisions effectively and thus to support human decision making with information technology.

To understand what is effective decision making, we introduce Simon's concept of procedural rationality. Simon (1976) states that it was only after the second world war that classical economic theory was supplemented by a perspective on economics that was based on *procedural, or bounded, rationality*. Classical economic theory rests on two fundamental assumptions. One, the economic actor has a well-defined particular goal. Two, the economic actor is *substantively rational*, which by definition stipulates that the rationality of the behavior of the actor depends on one aspect only: his or her goal.

Procedural rationality assumes that the concept of rationality is synonymous with *the peculiar thinking process called reasoning* (James, 1890). According to Simon (1976), behavior is *procedurally rational* when it is the outcome of appropriate deliberation. Accepting *procedural rational* behavior therefore makes the process of problem solving, or decision making, not a theory of best solutions, of *substantive rationality*, but a theory of efficient activities, i.e. to find good, or accepted, solutions (Simon, 1976).

The challenge now becomes to support the capriciousness of decision making, or problem solving, by supporting N decision makers with M perceptions, P constraints and Q goals. In this support we distinguish the process, or method from tools.

Sol (1982) presents simulation as a process, i.e. a method of inquiry, and advocates that simulation is the preferred method of inquiry for ill-structured problems. Simulation is defined as the process of designing a model of a real system and conducting experiments with this model for the purpose of either understanding the behavior of the system or of evaluating various strategies for its operation (Shannon, 1975). Simulation is thus a process to be supported by tools, i.e. decision support systems.

A leading question in the development of such decision support systems concerns the effectiveness of these systems. Keen and Sol (2005) argue that this effectiveness can be expressed by a combination of three Us: *usefulness, usability* and *usage*. The usefulness of decision support tools expresses the value they add to the deci-

sion making process. Usability expresses the mesh between people, processes and technologies, and usage expresses the flexibility, adaptivity and suitability of DSSs for organizational, technical, or social context. According to Keen and Sol (2005), traditional decision support tools do not place equal emphasis on the three Us: substantive rationality underlies their design.

Stressing all three Us equally results in the concept of decision support *studios*, *suites* and *services* (Keen and Sol, 2005). A suite is a well chosen set of services and recipes for inter-connectivity; a decision support suite is thus a chosen set of services and recipes to support a decision making process. A studio is a (virtual) environment in which suites are deployed, e.g. a group decision room or a web-portal.

We argue that the concept of a suite is explicitly related to the *service oriented society* we live in. The *service oriented computing (SOC)* paradigm, or *service oriented architecture (SOA)*, that underlies the design of modern information systems, represents this society in the design of information systems.

We now introduce the research question addressed in this thesis, which is based on theories of studio based decision making and service oriented software engineering.

Research question: Can we create a simulation suite for decision makers that supports a studio-based decision process and improves their effectiveness when solving ill-structured, multidisciplinary problems?

A scientific inquiry may best be illustrated as following a particular *process* or *strategy* in which a set of *research instruments* are employed and which is guided by the researchers using an underlying *research philosophy*.

Although philosophers from both the active and passive schools of thought have formed an underlying philosophy for organizational and information system research, we postulate that systems engineering is a subjective human creation and as such base our research on realistic activism, or revolutionary conventionalism.

March and Smith (1995) present two strategies, or paradigms, for the design of an information system: the *behavioral science* paradigm and the *design science* paradigm. The behavioral science paradigm seeks to develop and justify theories, i.e. principles and laws, that explain or predict organizational and human phenomena surrounding the analysis, design, implementation, management and use of information systems (Delone and McLean, 1992, 2003; Seddon, 1997). The design science paradigm has its roots in engineering and the sciences of the artificial (Simon, 1996). It is fundamentally a problem-solving paradigm.

We agree with the arguments of Hevner et al. (2004), who argue that design science and behavioral science should be engaged in a complementary research paradigm, and postulate such a complementary, explorative research strategy for this research. In this explorative strategy, behavioral science addresses research through

the development and justification of theories that explain or predict phenomena related to an identified business need. Design science addresses research through the building and evaluation of artifacts designed to meet the identified business need.

The outline of this research reflects the explorative research strategy. Two explorative case studies are presented in chapter 2. In this chapter hypotheses are conclusively refined to complete the initial phase of this research.

The concepts and theories of *object-oriented* system design are introduced in chapter 3, which is concluded with a set of principles for object-oriented systems design. In terms of the design science strategy, the object-oriented theories form the constructs; the principles form the methods.

We introduce several theories of *modeling* and *simulation* in chapter 4. The specification of the time-dependent behavior of a simulation model, i.e. the formalism of a simulation model, forms the main concept discussed in this chapter. We follow Vangheluwe and de Lara (2002), who argue that it is desirable to express the behavior of a model as a function of multiple formalisms. We conclude this chapter with a set of requirements for a simulation suite.

The actual design, or instantiation, is presented in chapter 5. Here we present our contribution to the field of systems engineering with the introduction of a distributed Java-based Simulation Object Library (DSOL), and we discuss the requirements, architecture and implementation of the DSOL suite. The suite consists of several services among which the simulation core service, an asynchronous event service, a logging service and a 2D and 3D animation service. The DSOL suite is published under an open source license and can be downloaded from <http://www.simulation.tudelft.nl>.

Verification and expert validation of DSOL is presented in chapter 6. The verification of the suite is mainly based on the implementation of a set of comparisons specified by Breitenecker (2004).

In chapters 7, named *Emulation with DSOL* and 8, named *Flight scheduling at KLM* we present a validation of our hypotheses that DSOL contributes to more effective decision support, based on the 3 Us. In both chapters real-life case studies are presented in which we explicitly focus on one or more of the Us.

We present our conclusions and explore potential future research in chapter 9. We conclude that by using DSOL *usability* is improved through the distributed web-based access, the separation of the modeling environment and experimentation environment, and the introduction of state-of-the-art software engineering tools. *Usefulness* of the decision making process is improved through the availability of several formalisms and well documented interfaces for domain specific libraries. *Usage* is improved and supported through the intentional absence of proprietary licenses, the absence of concealed parts in the architecture and a strong focus on

the support of multiple actors.

We finish this thesis with the more general conclusion that a service based approach to software engineering has enabled us to design and specify a full featured simulation suite, i.e. the DSOL suite. We strongly believe that DSOL, due to its open source license, its current user community and its well documented and open project management, will survive and go on to 'outlive' this particular Ph.D. thesis trajectory.

SAMENVATTING

De belangrijkste boodschap die in dit proefschrift, *de DSOL simulatie suite*, wordt uitgesproken is dat een nieuwe kijk op, ofwel een nieuw paradigma voor, het ontwerpen van besluitvorming ondersteunende informatiesystemen meer dan wenselijk is. Om te begrijpen hoe slecht gestructureerde besluiten effectief kunnen worden ondersteund introduceren we het concept van *procedurele rationaliteit*.

Simon (1976) beschrijft dat het tot na de tweede wereldoorlog heeft geduurd voordat klassieke economische theorieën werden aangevuld met een perspectief van de beperkte, ofwel procedurele, rationaliteit. Klassieke economische theorieën rusten op twee belangrijke peilers. Op de eerste plaats gaan deze theorieën uit van een economische actor met een duidelijk, eenduidig gedefinieerd doel. Op de tweede plaats gaan deze theorieën uit van een actor die substantieel rationeel handelt, hetgeen impliceert dat een actor slechts één doel nastreeft: zijn eigen.

Procedurele rationaliteit veronderstelt dat rationaliteit synoniem is met het proces van redeneren (James, 1890). Simon (1976) stelt dat gedrag procedureel rationeel is wanneer het resultaat is van een gedegen overleg proces. Procedurele rationaliteit maakt het proces van probleem oplossen een proces van zoeken naar een geaccepteerde oplossing in plaats van naar de beste oplossing. Een optimale oplossing bestaat dan ook niet; we moeten zoeken naar haalbare, ofwel reële oplossingen. We richten ons in dit proefschrift op het ondersteunen van besluitvorming in slecht gestructureerde problemen. Problemen zijn over het algemeen slecht gestructureerd wanneer ze zowel inhoudelijk, bijvoorbeeld technisch, als organisationeel, bijvoorbeeld door het groot aantal betrokken actoren, complex zijn. De vraag luidt nu hoe de grilligheid van deze besluitvormingsprocessen te ondersteunen. Hoe kunnen we met andere woorden N besluitvormers met M percepties, P randvoorwaarden en Q doelstellingen ondersteunen? Het is hierbij relevant het proces van ondersteunen te scheiden van de hulpmiddelen, ofwel tools, die aangeboden worden.

Sol (1982) beargumenteert dat simulatie een proces, een aanpak, ofwel een methode van probleem oplossen is; hij stelt verder dat simulatie de verkozen aanpak is wanneer we te maken hebben met het ondersteunen van besluitvorming van slecht gestructureerde problemen. We definiëren simulatie volgens Shannon (1975) als het proces van het ontwerpen van een model van een deel van de echte wereld met het oogmerk met dit model te experimenteren. Het doel van simulatie is ofwel

de werking van een bestaand systeem te leren begrijpen ofwel nieuwe strategieën voor besturing te evalueren (Shannon, 1975). Simulatie is dus een proces waarvoor tools, ofwel hulpmiddelen, ontworpen worden.

Een belangrijke vraag in de ontwikkeling van dergelijke tools is hoe effectief ze zijn. Hoe effectief is met andere woorden een simulatie tool, of omgeving, in vergelijking met zijn concurrenten? Keen and Sol (2005) stellen dat deze effectiviteit kan worden uitgedrukt als een functie van 3U's: usefulness, usability en usage. De usefulness, ofwel bruikbaarheid, van een besluitvormings ondersteunend informatiesysteem drukt de waarde uit die een dergelijk systeem toegevoegd aan het daadwerkelijke proces van besluitvorming. Is een systeem met andere woorden wel bruikbaar voor het type probleem dat opgelost dient te worden? Usability, ofwel gebruiksvriendelijkheid, drukt kwaliteit van de interactie tussen mensen, processen en technologieën uit. Usage, ofwel gebruik, drukt de flexibiliteit, adaptiviteit en sustainabiliteit van een dergelijk informatiesysteem in haar organisationele context uit.

Keen and Sol (2005) beargumenteren dat er in het ontwerp van traditionele besluitvormings ondersteunende informatiesystemen niet aan iedere U evenveel aandacht is geschonken. Waar sommige systemen duidelijk focussen op gebruiksvriendelijkheid, ligt de aandacht bij andere op bruikbaarheid. Het onderliggende probleem is dat substantiële en niet procedurele rationaliteit vaak de basis voor ontwerp is. Keen and Sol (2005) komen, als gevolg van het benadrukken van alle U's, met een studio concept voor het ondersteunen van besluitvorming. Een studio is een (virtuele) omgeving waarbinnen suites zijn gedeployed. Voorbeelden van studio's zijn een group decision room of een internet portal. Een suite is een geselecteerde set van services en recepten voor inter-connectiviteit die de studio vullen. Een service is gedefinieerd als de specificatie van een objectgeoriënteerde (sub)systeem dat een coherente set van functionaliteit via een of meerdere interfaces, of contracten aanbiedt.

We beargumenteren dat het concept van een suite, ofwel een set of services, direct gerelateerd dient te worden aan de service georiënteerde maatschappij waarin we leven. Deze maatschappij zien we in het ontwikkelen van informatiesystemen terug in paradigma's zoals *service georiënteerde computing* en *service georiënteerde architecturen*. Met dit in het achterhoofd komen we tot de volgende vraag die we in dit proefschrift pogen te beantwoorden.

Onderzoeksvraag: Kunnen we een simulatie suite ontwikkelen voor besluitvormers die een studio gebaseerde aanpak van besluitvorming ondersteunt en hen hierdoor in staat stelt effectiever slecht gestructureerde problemen op te lossen?

Om te komen tot het beantwoorden van bovenstaande onderzoeksvraag is specifieke onderzoeks aanpak doorlopen waarin verschillende onderzoeksinstrumenten

zijn toegepast, en die een onderliggende wetenschapsfilosofie reflecteert. Beginnend met de onderzoeksfilosofie zien we dat zowel de activistische als de passivistische school ten grondslag liggen aan de ontwikkeling van informatiesystemen. Wij postuleren in dit onderzoek een realistisch activistische filosofie omdat systeemkunde, ofwel systems engineering, hierin als een subjectieve menselijke activiteit wordt gezien.

March and Smith (1995) presenteren vervolgens een tweetal aanpakken voor het ontwerp van informatiesystemen: de *gedragwetenschappelijke* en de *ontwerpwetenschappelijke* aanpak. De gedragwetenschappelijke aanpak tracht theorieën over organisationele of menselijke fenomenen rondom het gebruik van informatiesystemen te verklaren of voorspellen (Delone and McLean, 1992, 2003; Seddon, 1997). De ontwerpwetenschappelijke aanpak kent zijn wortels in de engineering en de wetenschap van het artificiële (Simon, 1996); het is feitelijk een probleemoplossende aanpak.

Wij sluiten ons aan bij Hevner et al. (2004), die stellen dat een gedragwetenschappelijk en ontwerpwetenschappelijke aanpak deel zouden moeten uitmaken van een complementaire aanpak. We stellen een dergelijke exploratieve aanpak dan ook voor als leidraad voor dit onderzoek. In deze aanpak zullen de identificatie van het probleem en de validatie van het ontwerp volgens een gedragwetenschappelijke aanpak verlopen; het daadwerkelijke ontwerp van de suite zal volgens een ontwerpwetenschappelijke aanpak geschieden.

De opbouw van dit proefschrift weerspiegelt deze exploratieve aanpak. Twee exploratieve case studies worden beschreven in hoofdstuk 2. Aan het eind van dit hoofdstuk worden de onderzoeksvragen verfijnd.

Concepten en theorieën rondom systeemkunde en systeemontwerp staan beschreven in hoofdstuk 3. Hier wordt een duidelijke slag gemaakt naar het ontwerp proces. Dit hoofdstuk beschrijft objectoriëntatie als een wijze van systeem beschrijven en presenteert een tiental principes voor het object georiënteerd systeemontwerp.

We presenteren de concepten rondom simulatie in hoofdstuk 4. De meta-modellen voor de specificatie van het gedrag van simulatie modellen, ookwel formalismen genaamd, staan hierin centraal. Hoofdstuk 4 borduurt voort op de ideeën van Vangheluwe and de Lara (2002) die stellen dat het vaak wenselijk is een model als functie van meerdere formalismen te conceptualiseren en dus te specificeren. In een model worden dan bijvoorbeeld zowel discrete events, bijvoorbeeld aankomende passagiers, als continue functies, bijvoorbeeld afremmende voertuigen, gespecificeerd. Vandaar ook de subtitel van dit proefschrift: *het ondersteunen van multi-formalisme simulatie*.

De waarde van dit proefschrift komt tot uiting in het ontwerp dat beschreven is in hoofdstuk 5. Hier presenteren wij onze bijdrage aan het domein der systeemkunde met de introductie van een gedistribueerde, Java-gebaseerde Simulatie

Object Library (DSOL). We bespreken in dit hoofdstuk de ontwerpeisen, de architectuur van DSOL en de uiteindelijke implementatie van deze architectuur. DSOL bestaat uit een negental services waaronder een simulatie service, een service voor gedistribueerde asynchrone communicatie, een service voor 2-dimensionale en 3-dimensionale animatie, een statistiek service, etc. DSOL is gepubliceerd onder een open source software licentie. Dit betekent grofweg dat iedereen mag en kan doen met de software zoals hem/haar betaamt. DSOL is gepubliceerd op de volgende internet site: <http://www.simulation.tudelft.nl>

De verificatie en expert validatie van DSOL staan beschreven in hoofdstuk 6. In hoofdstuk 7 en 8 presenteren we vervolgens de validatie dat DSOL daadwerkelijk bijdraagt aan meer effectieve besluitvorming. In deze hoofdstukken staan real-life case studies beschreven die uitgevoerd zijn bij Dycore, een organisatie in Breda die zich toelegt op de productie van betonnen vloeren, en bij de KLM. In beide case studies focussen we op een of meerdere specifieke U's en laten we zien hoe DSOL zich in het ondersteunen van besluitvorming verhoudt tot bestaande simulatie omgevingen.

We presenteren onze conclusies en aanbevelingen voor verder onderzoek in hoofdstuk 9. De belangrijkste conclusie is dat we met DSOL inderdaad een omgeving aanbieden die effectievere besluitvorming tot stand brengt. We concluderen dat we de Usability van een simulatie omgeving met DSOL daadwerkelijk hebben vergroot. De voornaamste redenen zijn het web georiënteerd karakter van DSOL, het scheiden van een modelleeromgeving van een experimenteeromgeving en de introductie van state-of-the-art software tools in het domein van de simulatie. De Usefulness is eveneens tastbaar toegenomen. De belangrijkste reden, is het multi-formalisten denken. In DSOL kunnen modellen in meerdere formalismen gespecificeerd worden; de conceptuele blauwdruk die besluitvormers van een probleem hebben kan één-op-één vertaald worden naar een DSOL simulatiemodel. Verder is DSOL een open omgeving waarin het gebruik van externe bibliotheken, door het gebruik van contracten, ookwel interfaces genaamd, goed ondersteund wordt. Usage is tot slot beter ondersteund door het ontbreken van een gesloten licentie, het ontbreken van afgesloten delen in de code en een sterke focus op het ondersteunen van meerdere actoren.

We besluiten dit proefschrift met een meer algemene conclusie dat een service georiënteerde kijk op software ontwikkeling ons in staat heeft gesteld in een beperkte tijd te komen tot het ontwerp, de implementatie en het testen van een volledige simulatie omgeving. We zijn er tot slot stellig van overtuigd dat DSOL, gezien de open source licentie, een aanwezige gebruikers gemeenschap en een gestandaardiseerde project management omgeving, de duur van deze promotie zal overleven.

ABOUT THE AUTHOR

Nederlands (Dutch)

Peter Jacobs is geboren op 8 mei 1975 te Utrecht. Hij is opgegroeid in Heerlen alwaar hij in 1993 zijn eindexamen Gymnasium deed aan het Bernardinus college. In 1993 begon hij aan de studie Chemische Technologie aan de Technische Universiteit Delft. Na een jaar als voorzitter van Delft's grootste studentenvereniging, vervolgde hij zijn studie aan de faculteit der Technische Bestuurskunde; aldaar ontving hij eind 2001 zijn ingenieursdiploma.

Gedurende deze Delftse jaren was Peter bestuurslid in verscheidene studenten organisaties en stond hij aan de voet van Javel b.v., een bedrijf dat zich toelegt op de ontwikkeling van geografische informatie systemen. Het onderwerp van zijn afstudeerstage, verricht in het iForce Ready Center van Sun Microsystems in Menlo Park, Californië (USA), was getiteld 'Distributed components in a visualization environment'. Peter ving 1 Januari 2002 aan met zijn promotie. Gedeeltes van dit werk zijn gepresenteerd in in San Diego (CA, USA, 2002), Bled (Slovenië, 2003), Sophia Antipolis (Frankrijk, 2003), Seoul (Korea, 2003), Hong Kong (China, 2003), Delft (Nederland, 2003), Cambridge (UK, 2004) and Washington (USA, 2004). Peter heeft gedurende zijn promotie meer dan 15 studenten begeleid in het behalen van hun ingenieursdiploma.

English (Engels)

Peter Jacobs was born on May 8, 1975 in Utrecht, but was raised in the far south of the Netherlands, in Heerlen. In 1993, he obtained his Gymnasium certificate at the St. Bernardinus college in Heerlen. He started his studies at Delft University of Technology at the faculty of Chemical Engineering. After being president of Delft's largest students' association, he decided to continue his studies at the faculty of Systems Engineering, Policy Analysis and Management; he earned a M.Sc. degree in October 2001. During his stay in Delft, he was on the committee of several student associations and started Javel, a business focused at the development of geographical information systems. The topic of his Master's thesis, written in the iForce Ready Center of Sun Microsystems in Menlo Park, California (USA), was titled 'Distributed components in a visualization environment'. He started his Ph.D. research on January 1, 2002. The result of this research is presented in

this thesis. Part of the work was presented in San Diego (CA, USA, 2002), Bled (Slovenia, 2003), Sophia Antipolis (France, 2003), Seoul (Korea, 2003), Hong Kong (China, 2003), Delft (Netherlands, 2003), Cambridge (UK, 2004) and Washington (USA, 2004). Peter has supervised more than 15 Ms.S students throughout his Ph.D. trajectory.