An agent based architecture for constructing Interactive Simulation Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. mr. P. F. van der Heijden
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Aula der Universiteit
op donderdag 9 december 2004, te 10.00 uur

door

Zhiming Zhao

geboren te Jianhu, Jiangsu, P. R. of China

Promotiecommissie:

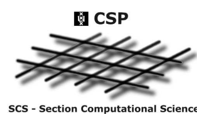
Promotor: prof. dr. P. M. A. Sloot

Co-promotor: dr. G. D. van Albada

Overige leden: prof. M. Boasson
prof. dr. F. M. T. Brazier
prof. dr. C. Sun
prof. dr. Z. Xu
dr. H. Afsarmanesh
dr. M. Bubak

Faculteit: Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The Work described in this thesis has been carried out at the Section Computational Science at University of Amsterdam, in the Advanced School for Computing and Imaging (ASCI) graduate school. It was financially supported by University of Amsterdam, and the European Union through contract number IST-2001-32243 under the CrossGrid project. This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). Part of this project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). Financial support was also received from ASCI.

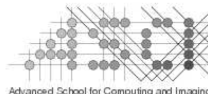


Section Computational Science

University of Amsterdam

The European CrossGrid project

The VL-e project
(Virtual Laboratory for e-Science)



The ASCI research school
Advanced School for Computing and Imaging

Copyright © 2004 by Zhiming Zhao. All rights reserved.

ISBN 90-6446-710-0
ASCI dissertation series number 108.
Printed by Ponsen & Looijen BV, Wageningen.
Author contact: zmzhao@idealinks.net

To Yan and to my parents.

Contents

1 Introduction **1**
- 1.1 Territory . . . 1
  - 1.1.1 Computer simulation . . . 1
  - 1.1.2 High performance computing . . . 3
  - 1.1.3 Scientific visualisation . . . 4
  - 1.1.4 Problem solving environments . . . 5
- 1.2 Towards an Interactive Simulation System . . . 6
  - 1.2.1 Requirements on the interconnection . . . 6
  - 1.2.2 Requirements on the code incorporation . . . 7
  - 1.2.3 Requirements on the Interaction module . . . 8
- 1.3 Modularity and integration . . . 8
  - 1.3.1 Middleware and interoperability . . . 8
  - 1.3.2 Activity orchestration . . . 11
- 1.4 Human-system interaction . . . 12
- 1.5 Real-time interaction . . . 13
  - 1.5.1 Performance and service quality . . . 13
  - 1.5.2 Time management . . . 13
- 1.6 Engineering methodologies . . . 14
  - 1.6.1 Software Components and ISSs . . . 15
  - 1.6.2 Agent technology and ISSs . . . 16
- 1.7 Summary . . . 18
- 1.8 Problem statement . . . 19
- 1.9 Thesis organisation . . . 20

2 An agent based component architecture **23**
- 2.1 Introduction . . . 23
- 2.2 Interactive Simulation System Conductor . . . 24
  - 2.2.1 Modules as reusable components . . . 24
  - 2.2.2 Basic architecture . . . 24
- 2.3 Agent based design . . . 25
  - 2.3.1 Agent definition . . . 25
  - 2.3.2 Activity control . . . 27
  - 2.3.3 Performance considerations . . . 27
- 2.4 Constructing interactive simulation systems . . . 27

2.4.1 Composing an ISS . . . 27
2.4.2 Run-time framework . . . 28
2.5 Summary . . . 29

3 Agent based activity orchestration **31**
3.1 An ISS as a multiple Module Agents system . . . 31
3.2 Inherent functionality: component capability . . . 32
3.2.1 Basic model . . . 32
3.2.2 Capability modelling for the human interaction involved components . . . 34
3.3 Interaction: story and scenarios . . . 35
3.3.1 Place transition net . . . 36
3.3.2 Scenario representation . . . 37
3.3.3 Transitions and actions . . . 38
3.3.4 Story: a scenario-net instance . . . 39
3.4 World model . . . 40
3.4.1 Basic structure . . . 40
3.4.2 Perception and uncertain belief of the agent world . . . 40
3.5 Controller . . . 42
3.5.1 Collecting observations . . . 42
3.5.2 Action execution control . . . 42
3.6 Story execution . . . 43
3.6.1 Basic paradigm: distributed scenario execution . . . 43
3.6.2 Hierarchical execution paradigm . . . 46
3.6.3 Centralised coordinator paradigm . . . 47
3.6.4 Scenario switch and execution paradigm selection . . . 48
3.6.5 Handling run-time exceptions . . . 48
3.7 Summary . . . 49

4 Implementation and performance analysis **51**
4.1 Communication agents . . . 51
4.1.1 Data object manager . . . 51
4.1.2 Distribution manager . . . 52
4.1.3 Events and action execution . . . 53
4.2 Module Agents . . . 53
4.3 Putting it all together . . . 53
4.3.1 Current implementation . . . 53
4.3.2 Actor and Conductor . . . 54
4.3.3 Capability and story descriptions . . . 55
4.3.4 Run-time configuration files . . . 55
4.4 Performance analysis . . . 56
4.4.1 Example components and the test bed . . . 56
4.4.2 Delay for remote updating shared objects . . . 57
4.4.3 Location of the RTI execution . . . 58

4.4.4 Remotely updating objects to multiple Consumers 60
4.4.5 Message passing 60
4.4.6 Object model and update delay 62
4.4.7 Summary 63
4.5 Performance for action reasoning and story execution 63
4.5.1 Overall observations on the action reasoning 64
4.5.2 Overhead of the reasoning kernel 66
4.5.3 Reasoning complexity and delay 67
4.5.4 Brief comparison between execution paradigms 69
4.5.5 Summary 69
4.6 Discussion and conclusions 70
4.6.1 Evaluation 70
4.6.2 Conclusions 71

5 Rapid Prototyping of a surgical pre-operative planning environment 73
5.1 Introduction 73
5.1.1 Background 73
5.1.2 Goal of the chapter 75
5.2 From Legacy systems to reusable components 76
5.2.1 Basic steps 76
5.2.2 Legacy flow simulation and visualisation systems 77
5.2.3 Component 1: *C_Flow_Simulator* 78
5.2.4 Component 2: *C_Desktop_VRE* 80
5.2.5 Discussion 81
5.3 Coupling component instances 82
5.3.1 Basic analysis: roles and interactions 82
5.3.2 Making an interaction story 83
5.3.3 Executing an ISS 84
5.3.4 Asynchronous data update 85
5.4 Automatic tuning of service quality 87
5.4.1 Adaptable state update 87
5.4.2 Solutions in ISS-Conductor 88
5.4.3 An example: adaptable rate for exporting *Flow_Data* 88
5.5 Collaborative interaction in an ISS 89
5.5.1 Requirements 90
5.5.2 Basic support 91
5.6 Collaborative data analysis and decision making 92
5.6.1 User opinions and decision points 92
5.6.2 Collaboratively exploring data 93
5.6.3 Experimental results 94
5.7 Multiple instances of a scenario net 95
5.7.1 Scenario template and data class mapping 96
5.8 Summarising discussion 97
5.9 Conclusions 98

6 Towards an intelligent planning environment for interactive simulations **101**
6.1 Introduction . . . 101
6.2 A global picture . . . 103
6.2.1 Proposed functional subsystems . . . 103
6.2.2 Design requirements . . . 105
6.2.3 ISS-Studio and Grid environments . . . 105
6.2.4 In the context of a PSE framework . . . 106
6.3 Intelligent planning of ISS-Conductor based interactive simulations . 106
6.3.1 Describing experiment requirements . . . 107
6.3.2 Component searching . . . 108
6.3.3 Story generation . . . 110
6.3.4 Generating execution scripts . . . 110
6.4 Prototype and preliminary results . . . 110
6.4.1 A multi-agent based experiment planning environment . . . 110
6.4.2 Experimental results . . . 111
6.5 Discussion and conclusions . . . 113

7 Summary and discussion **115**
7.1 Summary . . . 115
7.2 Conclusions and discussion . . . 116
7.3 Future work . . . 118

References **123**

Nederlandse Samenvatting **143**

Publications **145**

Index **147**

Acknowledgments **149**

Chapter 1

Introduction

1.1 Territory

From their inception over thirty years ago, human-in-the-loop simulations, also called interactive simulations, have become an increasingly important paradigm in a wide spectrum of applications, such as hardware design [1], industrial control [2], and special training for aerospace or battlefield [3, 4]. Interactive Simulation Systems (ISS) can significantly improve the efficiency of design verification, decision-making, and training. Allowing human users to manipulate simulation models and steer their execution at run time, ISSs are essential to realise Problem Solving Environments (PSE) for studying complex problems that are difficult to investigate using conventional methodologies.

The construction of ISSs is highly interdisciplinary; besides a profound knowledge of the application area, it involves the domains of modelling and simulation, scientific visualisation, human computer interaction, distributed computing and system engineering. The realisation of different development issues is often complex and time consuming. In scientific research, such development complexity critically hampers the productivity of ISSs; when a scientist explores a complex problem, he has to spend much of his effort on various implementation issues, instead of on the investigation of the experiment itself. A layered framework for developing ISSs is crucial to hide the underlying development issues from scientists and allow them to focus on the high-level behaviour of the system.

In this thesis we investigate a solution to the complexity issues in ISSs based on the separation of application logic control and system functionality. We demonstrate that this solution simplifies the development of ISSs and allows a scientist to quickly adapt a system to his needs in a rapid prototyping approach. In order to obtain a full view, we first take a short tour of the principal fields involved.

1.1.1 Computer simulation

Computer simulation refers to the process of building and operating models on computers for the purpose of gaining more insight into a system. It has become an impor-

tant methodology to complement normal lab experiments when it is too expensive, e.g. car crash procedures [5], or difficult to perform, e.g. nuclear reaction [6]. Although the systems being simulated may differ from each other, at an abstract level, all simulation experiments can be described in a similar manner, which mainly includes four iterative steps: building a computable model, validating the model, simulating the system and analysing experimental results.

Building a model for a system is to represent the system in a simple but sufficiently detailed way. A model need not include all the details of the real system, but it should contain those salient features, which permit us to draw valid conclusions about the actual system [7]. Using an artificial model to predict the behaviour of an actual system is a main goal of computer simulations, which implies that the model itself should be accurate enough and that it can be computed on an acceptably short time scale.

The validation of the model is based on doing simulation experiments and analysing the experimental results, where possible by comparing the results to real life experiments and observations. Fig. 1.1 shows general functional components of a simulator which the model shows some kind of evolution, e.g. according to a simulated time or of a control parameter that is changed in the course of the real time. Simulation experiments are often computationally demanding, for instance, the computation of a car crash simulator can take days to achieve certain accuracy [5], which makes the model validation very time consuming. The need to shorten the validation and execution time motivate a number of research subjects, such as high performance computing, scientific visualisation and human-in-the-loop of simulation.

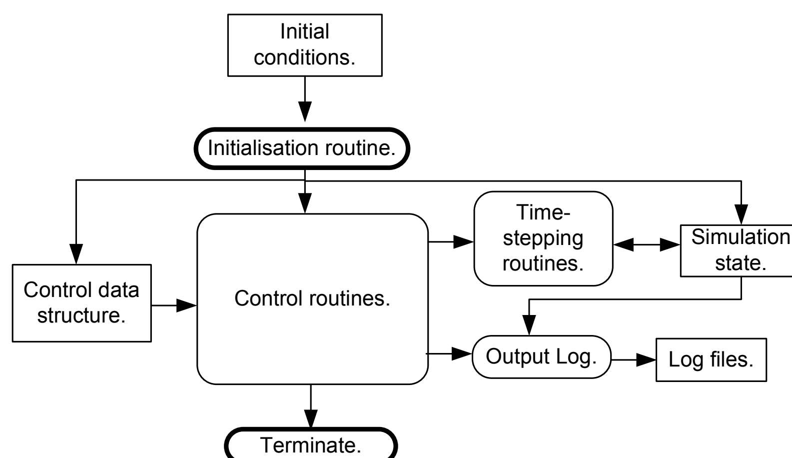


Figure 1.1: Functional components and the data flow in a simulator. The time stepping routines define the actual behaviour of the simulator, and the control routines define the experiment performed on the simulation.

1.1.2 High performance computing

Although the available computing power has increased enormously over the past decade (see e.g. Fig. 1.2), the demands for a further increase have not diminished. There are many good reasons for it. As the available computing power increases, more and more important problems just become feasible; for instance, a model with 2000 elements is adequate to advance the understanding of fluid dynamics 10 years ago, but now the models often contain more than 10^6 elements [8, 9]. But for many interesting problems, even a small increase in problem size or complexity demands a large increase in computing power, e.g. predicating weather for a longer time scale [10]. Apart from optimising the algorithms for doing the computation, the field of High Performance Computing (HPC) strives to make the highest attainable computing power accessible to the simulation researchers.

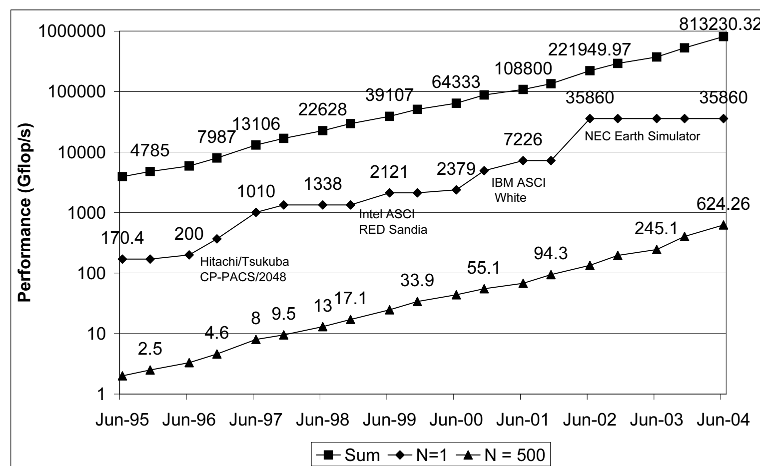


Figure 1.2: Computing power increases in the past decade. The figure shows the fastest (N=1) and the slowest (N=500) computer in the list, and the total performance of all computers in the list. The original information is from the website http://www.top500.org.

One way to gain better performance is to decompose a simulation model into a number of smaller sub-domains, and to compute them in parallel. To obtain real performance improvements, the parallisation needs to take a number of issues into account such as the quality of the domain decomposition, load balance between tasks, and communication efficiency. Besides, many important simulation models do not allow a straightforward parallelisation; the research subject of Parallel Discrete Event Simulation (PDES) is a typical example.

Recently, another attempt to gain more computing power is being developed, that is to horizontally interconnect available computing elements from multiple organisations in *Computational Grids* and share them among a defined group, called a Virtual Organisation (VO). The realisation of this ambition requires the resolution of a number of fundamental issues: resource discovery and allocation, execution monitoring and fault tolerance, and security controls. In turn, this novel execution environment also demands changes in the simulation model and its execution.

1.1.3 Scientific visualisation

In order to validate a simulation model or to explore the parameter space of a validated model, the developers have to study the data generated by the simulation experiments and compare them with the actual behaviour observed in the real systems. For complex models, the data can be multi-dimensional and large in volume, e.g. a blood flow simulator can generate more than 100 Mbytes for a full size abdominal aorta per time step [11]. Intuitively presenting data can essentially help researchers to digest the information in the data and to gain deeper insight into the problem. The process of mapping large quantities of data to the intuitive symbols that are perceivable for human senses, in particular vision, is called *scientific visualisation*.
The visualisation of data requires a number of processing stages, which in general include: pre-selecting relevant information from raw data, designing representations for the information, mapping the representation to intuitive primitives and rendering the primitives onto certain devices. Data is passed in a pipeline scheme between procedures; Fig. 1.3 shows a general data flow diagram. The systems that enable the entire pipeline and in particular support human users to interact with the rendered objects are called Data Exploration Environments (DEE) in [12]. Special devices, such as Virtual Reality (VR) systems, are often employed in DEEs for rendering and exploring complex and large-scale data.

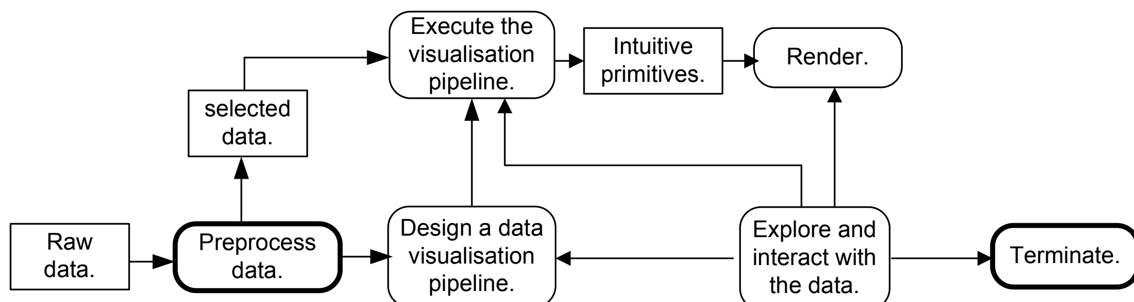


Figure 1.3: A general data flow diagram of visualisation systems. The procedures in a visualisation pipeline constitute the core of the system. The generation of intuitive primitives for rendering, and the user interaction that controls the execution of pipeline can be performed in different machines.

Due to the computational cost of data processing and visualisation, simulation experiments and the presentation of results are often separated in time. Static data is then the only way to pass information from one to the other. This requires additional investments for storing massive simulation results, especially when the simulation experiments are time-dependent. More importantly, it limits the efficiency for studying sensitive regions of the parameter space of the model when each configuration of the parameters requires a separate execution of the model. Yet, when the simulation itself can not generate meaningful results in a sufficiently short time scale compared to the cost for transmitting and viewing the volume of static data, little can be done to improve these shortcomings. With the continuing increase of the computing power,

the price of hardware coming down and the bandwidth of network increasing, it becomes feasible more often to include a real-time simulator into a visualisation as live data source, which motivates the work on integrating simulation and interactive visualisation.

1.1.4 Problem solving environments

Problem Solving Environments (PSE)s are integrated software environments that *provide tools and utilities necessary for solving a target domain of problems* [13]. PSEs couple different types of resources and computational technologies both horizontally and vertically and allow scientists to tackle the scientific problems at a high-level of abstraction [14]. Horizontally, a PSE provides gluing mechanisms for reusable software resources, e.g. simulators, visualisation and data analysis utilities, and allows scientists to build a new computing system by assembling these resources instead of developing new software. Vertically, a PSE provides hierarchical schemes to organise the computational knowledge involved in different types of resources and allows scientists to work on a given level without being experts on all the others, e.g. a simulation model developer does not need to be an expert in scientific visualisation.
PSEs were originally proposed in the early 1960s, but due to the strong dependence on computing power, they have only been successfully realised after the significant progress was made in HPC. After 1990, a large number of special purpose PSEs have been prototyped and implemented, e.g. VLAM-G [15], SciRun [16] and CtCoq [17]. Depending on the guise that a PSE takes in the lifecycles of problem solving, a PSE has also been called differently: e.g. Scientific Portal [18], Virtual Laboratory [19] and Virtual Workbench [20]. In this thesis, we use the term PSE to cover them all.
The functionality of a PSE depends critically on the use of computer simulations and can be greatly enhanced by putting a human in their run-time loops. The challenges for performing experiments using human-in-the-loop simulation not only lie in the development of a suitable simulation system but also in the management of all types of, both static and dynamic, data information involved in the experiment, e.g. system requirements, simulation results, and experiment histories. An efficient support for managing information can also promote its reusability as resources for new experiments.
The information management can be supported by a number of technologies. Kaletas categories these technologies from four perspectives [21, 22]. Data models and standards are the first one; describing data entities in a system using standard data models can not only facilitate the information sharing between different system users but also improve the efficiency for customising models for a new application. Successful standards include the Object Data Management Group (ODMG) standard [23] and Dublin Core Metadata standard [24]. The second one is from the perspective of managing distributed information. Distributed and federated information management provides flexible mechanisms to couple distributed databases for storing and accessing information; examples include Polar [25] and PEER [26]. Resource management is the third one; in a distributed computing environment, simulation, visualisation,

data and different types of tools are all considered as resources which can be deployed in customising specific run-time applications. A number of frameworks for resource management are developed in Grid computing environment, e.g. Open Grid Service Architecture based Data Access and Integration (OGSA-DAI) [27]. Finally, the support for information management is also provided in environments for managing workflow between distributed computing entities and the security control for the resource access [28, 29].

1.2 Towards an Interactive Simulation System

In general, a minimum ISS has three basic modules: simulation, visualisation and interaction. The simulation regularly computes and transfers data to the visualisation and interaction modules, and a human user can manipulate the simulation parameters through the interaction module, as shown in Fig. 1.4. In order to achieve a higher performance those modules often require dedicated hardware platforms and thus need to be run in a distributed environment.

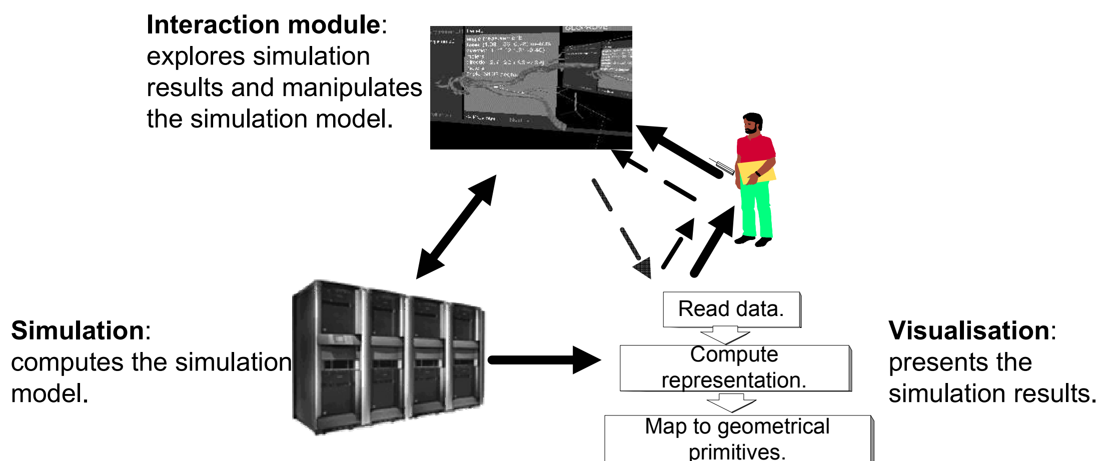


Figure 1.4: A basic configuration of ISSs. Solid lines depict the simulation loop, and the dash lines depict the visualisation loop.

An ISS is often constructed by integrating existing simulation and visualisation systems. Legacy simulators are likely to include verified implementations of algorithms that may be applicable to other problems; visualisation tools can be adaptable to work with different simulators for related domains. An efficient reuse of legacy assets will reduce both the development costs and risks [30, 31]. The integration between simulation and visualisation programs requires a number of changes in both, and the addition of a third module that allows a user to manipulate their run-time behaviour.

1.2.1 Requirements on the interconnection

Basic coupling issues include enabling interaction capabilities in simulation kernels [32, 33] and in visualisation procedures [34, 35], and communication between them

[36, 37]. The interconnection has to take into account a number of issues. The first one is the existing differences in the data representations at all levels. Sophisticated data specification, marshalling and interconnection techniques have recently become available in middleware such as Cactus [38]. The second issue is the very high performance needed for a timely rendering. The performance requirements on both the data connection between simulation and visualisation and on the visualisation itself can be reduced by using appropriate data selection techniques to limit the transmitted data to those immediately needed by the visualisation. This is a service that requires a detailed knowledge of the simulation and visualisation process. Therefore, it can only be provided at the application level. The third one is the co-ordination of the system execution. In simple cases, such as a simulation-monitoring system, e.g. the Jane framework [39], the dependencies between simulation and visualisation can be handled by a data stream which is controlled either by the simulation or by the user. In more complex cases where the user's feedback is to be included in the running loop of the simulation(s), the correct ordering and synchronisation of actions becomes even more difficult. Finally, the interconnection also has to take the support for information management into account, although the support itself might not be direct functionality of an ISS. Using standard data models to describe the information involved in interactive simulation based experiments allows a standalone system to support the information management, and coupling distributed ISS modules using a unified interface provides a standard way for the support system to gather run-time information and manage them in a distributed way.

1.2.2 Requirements on the code incorporation

With respect to a simulator, interaction means that the static data that controls the behaviour of the simulator may be changed during the execution, that the state of the simulated system may be modified, and that the control routine may be customised to be fit into the interaction context with the other modules. The first two kinds of changes must be influenced in a consistent manner throughout a (distributed) simulator program; both kinds can affect the stability and the convergence behaviour of the simulator. The latter changes affect the control, and possibly the initialisation routines, but should have little effect on the computing of simulation states, e.g. the time-stepping routines in Fig. 1.1.

The changes to the visualisation program are related to the fact that the data to be visualised now arrive as a stream from the simulator, which puts additional time-constraints on the visualisation process. Depending on the refresh rates of the simulation states and their volume and complexity, the visualisation process needs to be adaptable to maintain the synchronisation between the update of the visualised scenes and the evolution of the simulator. To assist the human user to digest the simulation states in their evolution context, a visualisation process also needs to complement the visualised objects with necessary temporal execution information.

1.2.3 Requirements on the Interaction module

The interaction module provides an interface for a human user to manipulate simulation settings and states, and ensures that the modifications take effect. The interdependencies between the user interaction and the data representation result in a tight coupling between the user interface and the visualisation module, which is especially true when the manipulations of the simulation states can only be done via visualised objects. The interface should provide the necessary support to the user for making these modifications. Apart from the human-system interaction, the interaction module also co-ordinates the activities of different modules. We can distinguish two extreme modes of control for the integrated systems: a strong mode where activities of each module are pre-specified as a total/partial order, and a weak mode where modules behave autonomously and interact with the others under limited constraints. Actual systems usually are a hybrid of these two extremes. Because of the strong dependencies on the specific application, part of the realisation of the interaction module is often fused with the control routines of simulation and visualisation modules.

Since the 1980s, ISSs have become an important subject in the community of modelling and simulation and high performance computing. Apart from the successful application in different problem domains, technical issues involved in constructing ISSs have also been extensively studied. In the remainder of this chapter, we will first discuss them from the perspectives of module coupling strategies, communication middleware, user interaction and engineering methodologies. After that, we address the scientific research question to be studied.

1.3 Modularity and integration

Coupling simulation and visualisation using a *Client-Server* paradigm [39] can keep the simulation and visualisation programs essentially unchanged and allows to run them in parallel over different computers. In the integration between simulation and visualisation, two levels of coupling can be roughly distinguished: interoperability and behaviour orchestration. The coupling can be realised in tight and loose schemes.

1.3.1 Middleware and interoperability

A typical tight-coupling communication mode is the use of high performance communication libraries such as CAVERN [40] to directly connect simulation and interactive visualisation modules. A loose-coupling solution can be realised by defining a standard interface for distributed modules and by interconnecting them using a run-time infrastructure. Tight coupling often achieve a good performance, but from an engineering point of view, it introduces strong dependencies between modules, and will decrease the system reusability and portability. In contrast, loose coupling allows

modules to function in a relatively independent manner; the replacement of a module will not require changes in the others. Since the 1990s, a number of software architectures and middlewares for distributed and interactive simulation systems have evolved. We use two examples to discuss how they support the transparent data access and the remote interoperability between distributed simulations.

SPLICE

SPLICE is developed at Hollandse Signaalapparaten B. V. (HSA) [41, 42] for large-scale distributed embedded systems. The architecture aims to reduce the complexity of the development of large, reactive distributed systems and to provide fault tolerance and real-time support. SPLICE couples distributed application processes by assigning each of them with a communication co-ordinator, called *agent*[*], as shown in Fig. 1.5.

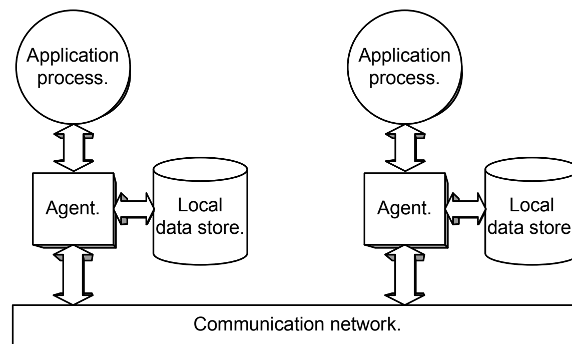


Figure 1.5: Basic components in SPLICE: application processes, each with an agent and a local data store.

In SPLICE, the data being exchanged between application processes is stored in the local data stores and is managed by agents. SPLICE distinguishes two types of data: *volatile* and *persistent*. *Persistent data* is always available for newly created processes, while volatile data is not. Agents exchange data using a publication/subscription mechanism. SPLICE is originally designed for real-time control systems and not for distributed simulation systems, but its agent-based communication mechanism does contribute a suitable paradigm for wrapping distributed simulation and visualisation programs and for interconnecting them [12].

High Level Architecture

High Level Architecture (HLA) is another example. It is proposed by the Department of Defence (DoD) of the U.S. as a standard architecture for interconnecting distributed defence simulators. HLA is a successor of two earlier protocols: Distributed Inter-

[*]We will have more discussion on this term in 1.6.2.

active Simulation (DIS)† for propagating states among distributed simulations, and Aggregation Level Simulation Protocol (ALSP) for synchronising simulators at run time [44] and for distributing events among them [45]. HLA enhances them by improving the support for interconnecting heterogeneous simulations and the scalability problems [43, 46, 47].
In HLA, modular components with a well-defined functionality and interface are envisioned as basic units, called *federates*, for building a simulation application, called a *federation*. In a federation, data properties are described as object models in which persistent data is called *objects* and messages for invoking remote activities are called *interactions*. Federate specific properties are described as simulation object models (SOM) which can be used to derive application specific data properties, called federation object models (FOM) [48]. Federates do not explicitly communicate with each other; instead, they are coupled using a *Run-Time Infrastructure* (RTI), via which federates subscribe to or publish the data classes that they can produce or consume. Apart from the data distribution, the RTI also serves the federates to update logical time and to manage global execution states. A federate invokes these services and reacts to the requests from the RTI through its local RTI library (libRTI). Fig 1.6 shows a logical structure of a federation. A process can contain multiple *libRTIs* to join multiple federations as different federates. The DoD's implementations adopt The ACE ORB (TAO) [49], an implementation of Real-time CORBA [50], as its basis‡.

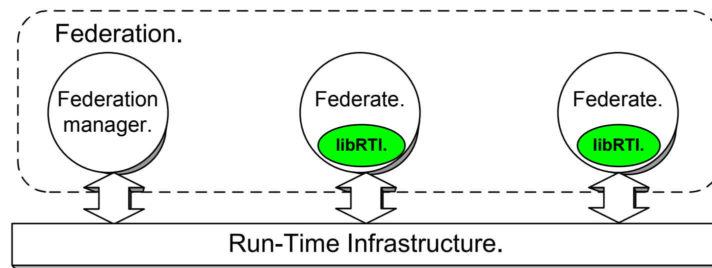


Figure 1.6: Distributed federates and the Runtime infrastructure (RTI).

Using the standard interface defined in HLA, simulation, visualisation and interaction modules can be wrapped as federates using libRTI. The data exchanged between system modules, e.g. simulation states and control data, are incorporated as data classes according to a federation object model, and can be remotely accessed by federates via the RTI. The dedicated services for distributed simulation also support the interoperability and interaction control between ISS modules, e.g. the time management services for the delivery of the control messages (*interactions*), and the data distribution management services for content-based data distribution. Although the initial development of HLA is for defence simulations, it was soon applied as a stan-

†DIS was developed in the SIMNET [43] project which was launched by the Defence Advanced Research Project Agency (DARPA) and the U.S. DoD in the early 1980s for constructing a shared synthetic military training environment.

‡After September 2002, commercial companies are also allowed to implement and realise the RTI software [51].

dard architecture in many other industrial and scientific simulation systems [52–54]. It also inspired a number of related research subjects, e.g. interconnecting federations [55], and enhancing the services for managing time [56] and for distributing data [57]. Later in this chapter we will discuss the time management in more detail.

1.3.2 Activity orchestration

It has been realised a decade ago that de-coupling the activity co-ordination from the system functionality can improve the flexibility to control the interaction of parallel and distributed systems. Co-ordination languages are an such effort [58]. Linda [59] is designed to complement computational language with a co-ordination module for managing the interaction between computing processes. Since it is at language level, the computation and co-ordination are bound at link stage. Workflow management systems (WFMS) [60] are a later example when significant progress has been achieved in middleware platforms. WFMS are originally developed for automating the interaction between business processes, but have also been applied to scientific applications, e.g. Condor [61] and VLAM-G [19].

Basically, a workflow management system works on a middleware with standardised interface and uses a centralised co-ordinator to orchestrate the system execution by scheduling the distribution of messages among system components. A workflow management system explicitly models interaction scenarios of the overall system, and manages resources which are required by them. In this section, we use VLAM-G as an example to discuss the solutions to these issues.

Process flow template and resource manager in VLAM-G

VLAM-G is a generic PSE framework, which provides hierarchical solutions to manage software and computing resources, and to allow scientists to utilise the resources to prototype and perform scientific experiments. In VLAM-G, the interaction scenarios in a scientific experiment are modelled as an abstract description of the processes, called Process Flow Template (PFT); an instance of a PFT is called a *topology* [15]. The flow between processes is described using data dependencies. At run time, the execution of a *topology* is scheduled and co-ordinated by a *scheduler*, e.g. allocating computing tasks, and establishing data flow between them. A PFT can have multiple topologies; each topology is handled by a separated scheduler.

VLAM-G provides a layered framework which allows domain experts and package developers to work collaboratively. It provides a visual environment to describe the PFT, and a user-friendly interface to monitor the execution of a topology. Currently, VLAM-G mainly support complex and data intensive experiments, e.g. hardware in the loop; the description of the interaction scenario is based on data flow.

1.4 Human-system interaction

Human-system interaction is another important issue in the ISS development. A large body of discussions on human-computer interaction and interface design can be found in the literature, such as on modelling interaction processes [62], on designing interfaces [63], and on human factors [64]. Compared to normal interactive graphical systems and interactive visualisation environments, ISSs pose additional concerns when designing their interaction capabilities because of the *distributed* and *heterogeneous* nature of the system.

The first concern is the *paradigm* of updating simulation states. Before performing a meaningful action on the system, a user needs to first digest the information provided by the system. The delay for the perception depends not only on the user's knowledge about the system, but also on the volume of the information presented by the system. Delays for generating and transferring data and visualising them in the interface are incurred before the user can see the presentation. When those delays can be negligible, e.g. in the case of simple simulations, the system modules can work synchronously with the user's interaction; for complex cases, an asynchronous mode is more practical. These two modes are also identified as *user driven* and *simulation driven* mode respectively in [12]. Due to the parallel and asynchronous relationship between simulation and visualisation modules, the system realisation demands explicit care in controlling the simulation contexts. This is because when the simulation kernel receives an action request, the context for the request is often in the past of the current state of the simulation.

The second concern is the *manipulation* of simulation models, which can range from only accessing and exploring the simulation results to modifying the simulation model at run time. Hurrion [65] identified them as three levels: *basic operations* that change parameters of the simulation, *priority interactions* that schedule the execution of the operations, and *algorithm interactions* that change driving algorithms of the simulation model. In early systems, the limited capability of presenting information and supporting interaction restricted the freedom that a user could control the simulation. In the later ISSs complex interactions, such as refining 3-dimensional geometrical structures of the simulation, became feasible. In the system development, the support for users to accomplish the manipulation must also be addressed, because the users of an ISS may have different levels of domain knowledge and experiences.

The last one is the *portability* of the user interface. When an interface can only be presented on specific hardware, such as an immersive virtual reality device, it will be less portable than when it can be presented in a normal web browser. As we mentioned, special devices are often preferred for exploring complex and large-scale data information. But as computing intensive simulations will often last longer than a user can stay in the special hardware environment, a widely available interface to instantly access and monitor the simulation processes is also needed. They complement each other. The designer has to consider both the cost for providing a multiple access front end to the heterogeneous user interface and the characteristics of the

simulation kernel.

1.5 Real-time interaction

In ISSs, interaction with the simulation requires the system to respond in near *real time*. The term *real time* means that a system should not only be functionally correct (must produce its results correctly) but also temporally correct (act within specified time interval) [66]. Compared with the critical safety systems such as air traffic control and nuclear plant monitoring, the sense of real time in ISSs is *softer*, occasional delayed operations or error actions will not make a system absolutely unacceptable[§]. Generally, an ISS has to respond to the user's request and take its actions with an acceptable delay. Special purpose ISSs, such as defence simulations, have additional meaning for *real-time* that the evolution of the simulated world has to progress according to a referred time meter, such as *wall clock* time. In this section, we will briefly discuss two aspects of this issue: the service quality and the synchronisation between distributed simulations.

1.5.1 Performance and service quality

To realise real-time interactions, the system performance is critical; simulation and interactive visualisation modules must perform sufficiently fast to first satisfy the minimum requirement demanded by the user interaction and further to be adjustable to the desired time scale according to the evolution of the simulation states. Efficiently utilising the available computing resources can improve the performance, yet it is often necessary to optimise the system implementation itself. First of all, the improvements in the system performance can be obtained by employing custom technologies for data transmission and presentation, e.g. using multiple connections [37] or compression techniques [68] to transfer large volumes of data, or distributing visualisation modules to several computers and rendering them in a dedicated environment [69]. Secondly, the quality of the system services can be maximised by the users when they are allowed to make trade-offs between the resolution of the data presentation and the delay of the transmission and visualisation. Finally, services for optimising resource allocation include monitoring and balancing computation load, such as job migration [70], can improve the run-time performance of the entire system.

1.5.2 Time management

When an ISS comprises multiple simulations, as in defence simulation systems, each simulation must progress in accordance with the global evolution of the system in order to get correct overall behaviour. Causality conflicts occur when a simulation advances its time at a different rate than the other simulations expect. The execution

[§] The DoD's RTI does not support *hard* real-time distributed simulations [67].

of distributed modules must be co-ordinated: each simulation should treat its time correctly and the events should be interpreted by simulations in a correct order. Those issues are the concern of time management.

Protocols for synchronising distributed processes have been extensively studied in research concerned with PDES [71]. In PDES, the causality dependencies between simulation processes are dictated by their timestamps. Basically two categories of protocols are available: *conservative* and *optimistic*. Conservative protocols require that each process only processes the events with the minimum timestamp. In contrast, optimistic protocols allow simulation processes to evolve without waiting for the smallest time-stamped events, but employ additional mechanisms for detecting and recovering from the causality conflicts. The work in PDES primary focuses on performing DES in an as-fast-as-possible manner, but it has been successfully adopted in real-time interactive simulation systems. For instance, the DIS adopts a conservative Chandy/Misra/Bryant style null messages protocol to synchronise simulators, and in HLA, transparent interoperability between different types of simulations is supported [72].

In HLA, a federation execution manages its time issues using services for distributing messages and for granting time [73]. Messages can be delivered and received according to the logical time of the federation. HLA categorises different types of simulators, e.g. time driven, discrete event driven, real-time, or mixed, as four combinations: time regulated or not (for sending) and time constrained or not (for receiving), and transparently interconnects them. With the time granting services of the RTI, federates advance their local simulation time. The time-regulated federates can send messages with timestamps. In non-time constrained federates, the local time is granted immediately after being requested. In the time constrained federates, the local time of a federate can only be granted to the logical time which is smaller than the timestamps of all the messages being delivered to the destinations. There are three types of time-constrained federates: conservative event-driven, conservative time-stepped and optimistic. A conservative federate only processes a message when its local time has been granted to the time indicated by timestamp of that message. In conservative time stepped simulations, a *lookahead* value is set as the interval between time steps. Optimistic federates aggressively process messages and rollback when they receive a *straggler* with a smaller timestamp [74]. The RTI determines the Lower Bound Time Stamp (LBTS) for each federate.

1.6 Engineering methodologies

For a fully functional ISS, both the overall architecture of the system and that of the individual system modules tend to be complex. This introduces difficulties in the implementation, and in particular in keeping the system robust and easy to maintain. Engineering disciplines for tackling large problems provide a number of methods for managing complexity such as decomposition, abstraction and organisation [75]. Different software engineering methodologies, such as object oriented, component ori-

ented, and agent oriented engineering, brought contributions to designing ISSs.

1.6.1 Software Components and ISSs

Szyperski characterises software components as "*units of composition with contractually specified interfaces and explicit context dependencies only; they can be deployed independently and are subject to composition by third parties.*" [76]. Components encapsulate the implementation complexity of software into black boxes, and provide deployment level reusability. Generally, *interface* and *composition* are two basic concepts for building components and component-based systems. The interface describes the conditions under which the component can provide services and the precise nature of the services, it is intuitively viewed as a contract [77] between component developer and the potential customers. A component-based system is constructed by composing and assembling components using a framework which is often designed as a tiered architecture [78].

It has been realised by ISS developers that employing reusable building blocks can significantly improve the efficiency not only for the system development itself, but more importantly for the construction and composition of high-level, large-scale simulation models or for presentations. In the simulation environment, metaphors such as objects would be employed to abstract and represent the basic elements of physical models. ENVISION [79], JAAFAAR [80] and IMSAT [81] provide examples of this approach. In visualisation environments like IRIS explorer [82] the procedures for visualisation are formalised and packaged as a number of *modules*, which can be assembled to build pipelines to visualise a specific set of data. Although in these cases, the notion of *components* has not been explicitly used in the system architecture, the use of customisable building blocks that can be reused for composing specific applications does share common characteristics with software components [83]. In other systems, those *building blocks* are constructed using industrial standard component architectures: Java Beans are used to implement the model primitives in JISM [84], CORBA Components are used for wrapping simulations and for facilitating the control of computing processes [85, 86].

Industrial component architectures are mainly designed for the object systems in enterprise and business domains; explicit support for low latency communication, thus parallel interconnection between components, and in particular the parallel layout of the data structure is not addressed. More importantly the interoperability between different languages demanded by many simulation systems is not addressed in the implementation of those components. Novel component architectures suitable for high performance computing and interactive simulation systems are needed to reap the benefits of the engineering disciplines of component technology in the system construction.

Common Component Architecture

The Common Component Architecture (CCA) brings features for industrial software component architectures into high performance computing. Similar to normal industrial software components, the interface of a CCA component is modelled as a set of typed *ports*, which are described using a description language, called Scientific Interface Description Language (SIDL). At the deployment level, the composition between components is realised by connecting their ports, and the entry of the connection is defined in a special component called *driver*. The information about the connections is maintained in an object called component service by the framework. Since the integration between components and between components and the framework is implemented using ports, sophisticated flow control for the component activity has to be realised inside the components.
At run time, the components are instantiated by the framework, and each component obtains the information about the outside world through the component service object. Most of the component interactions are mediated through the framework. To improve the run-time performance, the CCA distinguishes the address spaces of components when binding them; direct invocations provided by underlying interfaces such as MPI or PVM are supported for the components in the same address space [87]. The CCA framework can support complex parallel computing by using different frameworks and special communication components.

HLA federates and components

Extending the architectures for interactive simulation and fitting them nicely to the concept of component-based engineering is another research subject. In [88], Radeski et al., stated that separating simulation logic from the integration interface of the RTI is an essential step to mate HLA federates with the component disciplines. Other researchers described mechanisms to combine HLA federates with CORBA Components [89] and Java beans [90]. To realise the explicit control of simulation logic, the SIMULTAAN Simulation Architecture (SSA) [91] employs a special federate called "scenario manager". Such a co-ordinator based mechanism is a common solution for controlling the task executions in workflow based systems [92, 93].

1.6.2 Agent technology and ISSs

Where component technology primarily addresses the problem of integration and interoperability in complex software systems, agent technology addresses the control of these systems. The Agent Oriented (AO) methodology complements the component method with knowledge related notions to manage system complexity [94]. The concept of *agents* originated in the mid-1950s as *a 'soft robot' living and doing its business within the computer's world* [95]. Nwana [96] identified two main strands in agent research. The first strand started about 1977, evolved from the field of Distributed Artificial Intelligence (DAI); its main research interests were in the theoretical perspectives of agents, e.g. deliberative activities, symbolic reasoning and

agent architectures. The second strand started about 1990, it mainly focuses on applying agents as an advanced technology for solving practical problems, e.g. agent oriented engineering and system modelling. Wooldridge distinguished three types of agent architectures: deliberative, reactive and hybrid [97]. The difference between the deliberative and reactive architectures is that the former incorporates a detailed and accurate symbolic description of the external world and uses sophisticated logic to reason about the activities, while the latter one only implements a stimulus-reaction scheme. Reactive architectures are easier to implement but lack a subtle reasoning capability. Hybrids of the two schemes are commonly used.

Agent technologies contribute to simulation based applications both a new modelling paradigm, as well as an intelligent solution to system development. Modelling a complex system as a multi-agent system captures the nature of the system behaviour in a bottom-up manner. Using agents to model and simulate a system, the domain is decomposed and mapped onto different roles of agents, and human-like behaviours, such as reasoning activity, are used to model the system behaviour. Successful examples include transportation systems [98], analysing air spaces [99] and social simulation [100]. Agent based simulation environments, such as SWARM [101], REPAST [102] and ASCAPE [103], are developed to facilitate the construction of simulations. As an intelligent solution, agent technologies have been reported in a large number of publications for implementing specific functions in interactive simulation systems, e.g. interaction support [104], probing information [105], co-ordinating distributed modules [106], facilitating complex system controls [107], and distributing data objects [108]. Besides these applications, agent based frameworks or middleware that can couple simulations and visualisation utilities and control their executions are also developed. One of the examples is the Bond agent environment [109].

Bond

In the Bond architecture, an agent is defined as a mobile object with a certain degree of intelligence for controlling its behaviour. The Bond framework is implemented in Java; the agents extend Java objects with communication support and reasoning. An agent has a model of the external world, and has an *agenda* containing its goals. The capabilities of the agent are represented by a hierarchical state transition graph. In the transition graph, strategies are associated with different states, and a strategy is basically a sequence of actions. The meta configurations for agent control are stored in a *blueprint* repository, which is accessible by the *Agent factory* to create agents. The *strategies* are also stored in a data base. Agents exchange messages using the XML [110] or KQML [111] formats and a globally shared tuplespace is used to enhance the message communications. Between hosts, communication engines provide underlying interconnection services. The global activities are controlled by a workflow management agent which co-operates with a performance monitoring agent. The Bond architecture has been successfully applied to implement and interconnect PDE

solvers [112].

1.7 Summary

The development of an ISS involves different issues: valid simulation and visualisation kernels, interoperability between distributed modules, and orchestration of the system behaviour. The use of advanced engineering methodologies to improve the *productivity* of developing interactive simulation systems actually started nearly a decade ago. The early work has addressed three main levels in the software architecture. At a middleware level, platforms like the DIS and HLA contribute a well-defined interface for supporting interoperability between distributed simulation components. At a simulation development level, reusable component or agent architectures, e.g. Java beans, CCA or Bond, and the engineering technologies supporting these architectures are used to construct systems components. And at an application-logic control level, interaction scenarios have been isolated from the simulations in a number of systems, e.g. SSA.
A suitable architecture for interactive simulation must provide the following support:

1. Wrapping legacy simulation and visualisation programs. Reusing existing mature simulation and visualisation kernels reduces development costs; the architecture necessarily provides interface to wrap legacy assets and to couple them using certain frameworks.

2. Interoperability between distributed system modules. A framework supporting transparent access and invocation of remote data and operations is essential to realise a loose-coupling scheme between system modules. Apart from it, services for distributed simulations, e.g. managing time and distributing data based on simulation context, are also needed when the system supports complex interaction.

3. Real-time interaction is a basic requirement for human-in-the-loop interaction, although hard real time is not required. The implementation of the architecture has to take the system performance into account.

4. Orchestration of system behaviour. An explicit and flexible control of the system behaviour allows an ISS to be customised for different scenarios, and can therefore efficiently shorten the life cycle of system prototyping. The orchestration mechanism has to not only provide a powerful description of possible interaction scenarios, but also an efficient paradigm to co-ordinate the execution.

A comparison of existing architectures is shown in Fig. 1.7. We can see, all the architectures in Fig. 1.7 can be used to wrap simulation and visualisation programs and to support interoperability between distributed system modules. Most of the architectures provide a description mechanism to specify the functionality of the components,

	Wrapping	Interoperability	Real time interaction	Interface description	Activity orchestration	Composition	Comments
HLA	Local RTI lib.	RTI.	Soft/with time management	Simulation Object Model.	**Implicit**: In the federate functionality.	Federation Object Model.	Suitable for interoperability between distributed simulation.
SSA	Based on HLA.	RTI, and RCI.	Soft /HLA based.	Simulation Object Model.	**Explicit**: In a centralised co-ordinator.	Simulation scenario.	Based on HLA, with multi-level communication, and a centralised scenario manager.
SPLICE	*Agents*.	*Agents*.	Hard.	Data.	**Implicit**: Inside the functionality of application processes and agents.	Data flow.	Suitable for hard real-time support, and distributed command control system.
CCA	Ports.	Framework	Soft.	Scientific IDL.	**Implicit**: Inside components and framework.	Data flow.	Suitable for high performance computing.
Bond	Bond agents.	Bond framework.	Poor.	Finite state machine.	**Implicit**: Inside Bond agents.	Blueprint (Finite state machine based).	A Java based implementation. With certain reasoning ability on behaviour control.

Figure 1.7: Summary of available architectures.

e.g. the SOM (Simulation Object Model) in HLA and the SIDL (Scientific Interface Description Language) in the CCA, and an integration mechanism for assembling the components and for realising their run-time binding. In those architectures, the interface specifications are basically used to promote the interoperability between components; an explicit layer for controlling overall interactions is not defined. In HLA, this often causes a tight coupling between simulation logic control and the federate implementation. In the CCA, a port-based interface is used to describe the interconnection structure between components. This mechanism can simplify the description of data and control flows, but the structure-based description fails to catch temporal behaviour aspects e.g. the concurrent activities and run-time conditions on the component connections, and the realisation of such cases will have to involve the code level development.

1.8 Problem statement

A clear de-coupling between application logic control and the inherent functionality is essential to enhance the reusability of the simulation assets and the adaptability of the system behaviour. Each of available architectures has its success stories in application. However, we have seen from the analysis none of them can both explicitly promote such separation and provide all necessary services for distributed simulation. They either realise the application logic control such as activity orchestration inside the constituent system components, or have limited capability to model hu-

man interactions at the system behaviour level. This leads to the statement of our research problem.
The main goal of the research is to *enhance existing architectures by providing a mechanism to separate application logic control from the inherent system functionality, so that they can utilise legacy simulation and visualisation programs to create the resources demanded by PSEs so that scientists can rapidly prototype an ISS and concentrate on deploying it in scientific experiments from the perspective of the high-level activities rather than the development of the system itself*. More specifically, we investigate the requirements on these issues at a system level, and propose a novel ISS architecture based on the state of the art of distributed simulation middleware and engineering technologies. We provide a proof of concept by defining an extension to an existing architecture: HLA.

1.9 Thesis organisation

In this thesis, we propose an agent-based architecture, called Interactive Simulation System Conductor (ISS-Conductor), for encapsulating the functionality of the legacy simulation or visualisation systems and for realising the interconnection between them. The thesis is organised as follows:

In chapter two, we introduce the architecture of ISS-Conductor, and briefly discuss its design issues. In the architecture, the component activities and system behaviour are orchestrated by a number of agents. In the knowledge base of the agents, the component capability and the application specific interaction constraints are described.

In the third chapter, we describe the design of ISS-Conductor, in particular the control mechanisms of the system behaviour. The modelling mechanisms of component capabilities and interaction scenarios, and three execution paradigms of agents are discussed.

The implementation details of ISS-Conductor are described in chapter four. The current implementation is on top of High Level Architecture and realises the reasoning functionality using Prolog. We also discuss the performance characteristics of the implementation.

In the fifth chapter, we use an application from biomedicine to demonstrate the main features of ISS-Conductor. We discuss the procedures to incorporate a legacy system into the ISS-Conductor architecture, and to deploy them into an interactive system. The discussion also includes agent based performance control and collaborative interaction support.

In the sixth chapter, we discuss the issues on one level higher: the feasibility of automated composition of ISS-Conductor based interactive simulation. We propose

an environment, called ISS-Studio, to facilitate the construction of ISS-Conductor compliant components, the assembling of interactive simulation systems, and the control of their execution. We focus on the issues related to semantic level component discovery and agent support for component assembling.

The final chapter summarises the research and discusses its future directions.

Chapter 2

An agent based component architecture

De-coupling application specific control from the functionality of a system is essential to improve the reusability and flexibility of its constituent components. Interactive Simulation System Conductor (ISS-Conductor) is an agent based architecture proposed to complement existing middlewares*.

2.1 Introduction

Separating application specific control from the basic data level operations requires mechanisms for both describing and orchestrating the high-level system behaviour. Data flow is the most popular mechanism to model system level interactions, e.g. in [16, 113]. Basically, there are two paradigms to orchestrate system behaviour: using a dedicated interaction co-ordinator, e.g. in workflow management systems [60], to schedule the control events between system modules, or incorporating each module as a standard machinery, which is able to autonomously interpret interaction constraints. The first paradigm has been successfully applied in a number of applications [114, 115]. However, in the case of ISSs, where the user drives the system interactions, such paradigm shows a number of shortcomings. First, the paradigm mostly deals with data flow based dependencies; to support sophisticated control for human-in-the-loop interactions, not only a suitable description mechanism is demanded, but also the monitoring on run-time states in each module is required for interpreting the system interaction constraints. When the number of system modules increases, the centralised control paradigm faces fault tolerance and scalability problems. Second, using a separate co-ordinator to orchestrate the system behaviour on the one hand can reduce the requirements on the code incorporation of simulation

*Parts of this chapter have been published in *Z. Zhao, R. G. Belleman, G. D. van Albada and P. M. A. Sloot*. "System integration for interactive simulation systems using intelligent agents", in the Proceedings of the 7th annual conference of the Advanced School for Computing and Imaging, May 2001. An extended version has also been submitted to international journal Concurrency: Practice and Experience.

modules, but on the other hand that will also introduce dependencies between the co-ordinator and the control interface of the system modules. Unlike from the work presented in [87,116], we propose a novel architecture based on the second paradigm.

The goal of ISS-Conductor is to provide a layered architecture for encapsulating the functionality and the run-time control of ISS modules, and for rapid prototyping of a system. The control of a system as complex as an ISS requires certain intelligence. Agent technologies provide a suitable approach to include control intelligence with the behaviour of a set of operations, therefore, we use them to interface the services provided by underlying ISS middleware, and to implement the orchestration of the system.
This chapter is organised as follows. First, we briefly introduce the ISS-Conductor architecture, and then describe its deployment in prototyping interactive simulation systems. Finally, we discuss the requirements and challenges of the development.

2.2 Interactive Simulation System Conductor

2.2.1 Modules as reusable components

In order to make the reuse of legacy ISSs really efficient and worthwhile, a degree of granularity for the components to be reused must be carefully chosen. The smallest components that could reasonably be considered are the individual routines, which would be packaged into a reusable library. Such libraries might be similar to existing mathematical and visualisation libraries, and would lose much of the sophistication of the carefully crafted simulation and visualisation modules, as the essence of the modules often lies in the interaction between the routines. The largest component that could be considered for reuse is the actual ISS as a whole. From the preceding description of its structure, it will be clear that an ISS usually cannot be adapted to a new application domain, or even to a new execution environment, without a major overhaul. A better granularity is at the module level: the principal modules of an ISS, e.g. simulation or interactive visualisation, are envisioned as basic reusable units for constructing ISSs. These units are encapsulated as *components*, in which the computing core from the principal module, e.g. the time-stepping routines in the simulator, would remain nearly unchanged.

2.2.2 Basic architecture

Originally, in designing ISS-Conductor, an approach was chosen where a relatively small control and I/O module would be added to a largely unchanged legacy code, with the aim of adapting the module's behaviour to the requirements of an ISS [4,5]. By keeping these foreign modules apart from the legacy control routines we could localise the application specific control and keep the major part of the legacy code

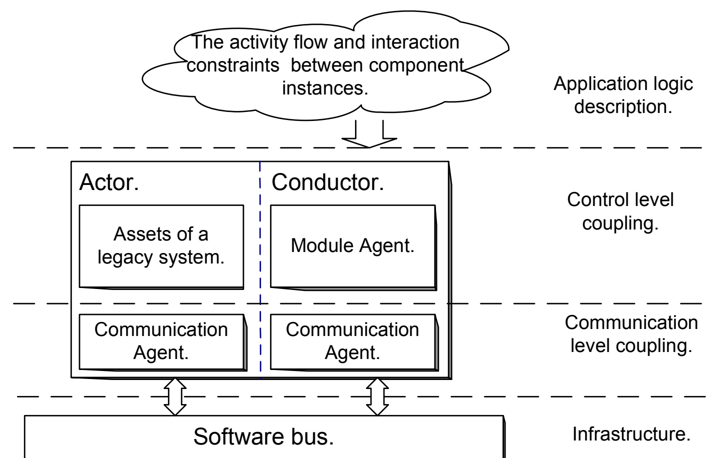


Figure 2.1: Basic architecture of an ISS-Conductor component.

unchanged. However, this approach allowed only limited control of the system behaviour. Therefore, the control routines of the legacy systems were also modified and so that they can be controlled by a foreign module. An ISS-Conductor component thus consists of two primary parts: an *Actor* for encapsulating the functionality of a legacy system and a *Conductor* for controlling the run-time activities of the *Actor*. The run-time integration between component instances is through a software bus.
Inside a component, both the *Actor* and the *Conductor* have a *Communication Agent (ComA)* which provides a uniform interface to exchange information with the software bus. In the *Actor*, the ComA wraps the computing core and data structures in the legacy code and provides an interface for remote access and invocation. In the *Conductor*, a control agent called a *Module Agent (MA)* provides the control intelligence for the behaviour of the component in a specific application context. Fig. 2.1 depicts the basic architecture.

2.3 Agent based design

2.3.1 Agent definition

ComAs are designed using a reactive architecture, as shown in the left part of Fig. 2.2. In the reflex architecture, sensors and effectors are interfaces for the agent to exchange information with the external world. The sensors listen to the external world, generate tasks for the recognised events and pass them to the task interpreter. The interpreter finds proper actions for the task in a task-action lookup table. Finally, the effectors carry out the actions. An important reason that we start with this architecture is because it is simple and extensible. For instance, when the task-action lookup table and the interpreter are complemented with a reasoning engine, the intelligence for action selection and execution will be immediately improved. Using it, agents can thus be constructed incrementally.

In the Actor, a ComA wraps the legacy system and interfaces it to the underlying communication middleware to realise the information exchange with the other ComAs. The functional components in the legacy systems are incorporated as activities in the ComA, and their associated control data are represented as data objects which can be accessed and manipulated remotely by the other components via the software bus framework. Since ComAs do not require sophisticated reasoning mechanisms for controlling their actions, they either pass the events observed from the external world to MAs or directly carry out the instructions sent from MAs. Fig. 2.3 shows an example of ComA in an Actor.

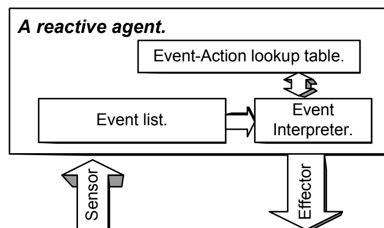


Figure 2.2: A simple agent kernel.

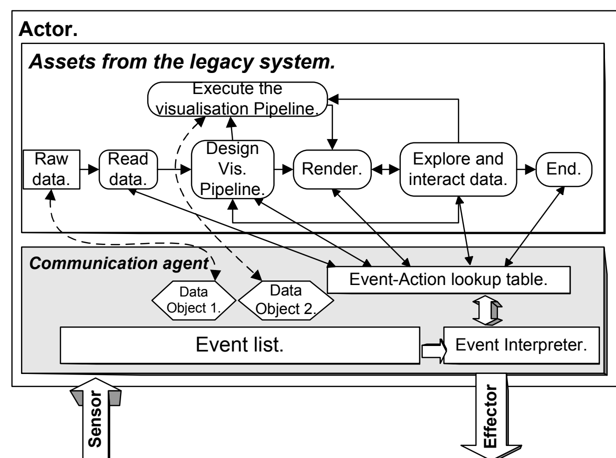


Figure 2.3: An Actor and its ComA.

An MA incorporates aspects of a deliberative agent. It employs a world model to track the changes of the external world and to obtain rational perceptions on the actual execution states of the other modules. Using the information supplied by the world model and the knowledge represented in the knowledge base, a reasoning engine is then used to find proper actions for the *Actor* to perform. Fig. 2.4 depicts a basic MA architecture.

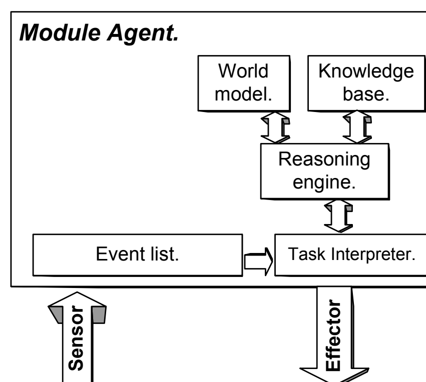


Figure 2.4: The basic architecture of MA.

2.3.2 Activity control

Inside a component, the action execution and the reasoning process are carried out by the Actor and the Conductor respectively. The capability of the simulation or visualisation system is described as knowledge of the MA in the Conductor, and its implementation is located in the Actor as a collection of actions and data objects. To take part in an ISS, each MA also receives a description of the interaction constraints and dependencies with the other components. Using these two types of knowledge, together with the information from the world model, an MA can then take a decision on the actions that the Actor should perform. After performing an action, the Actor reports the execution status to its MA while updates its world model. To keep the world model up to date, MAs also have to exchange their perceptions. The data objects that represent the control data or states in the legacy simulation or visualisation systems are maintained by the ComA in the Actor, and most of these objects only need to be accessible and shared among Actors. Depending on the level of detail of the knowledge representation, some of those objects are also needed by the Conductor of the component.

2.3.3 Performance considerations

As mentioned, adaptability and flexibility are often traded against performance in ISSs. In the ISS-Conductor architecture, performance is considered in three ways. First of all, the *Actor* retains the well-tuned computational kernels of the legacy implementation of the simulation and visualisation, and the interface for realising the parallelisation, such as MPI [119] or PVM [120]. Secondly, the implementation of ComA employs dedicated middleware for data distribution, which not only offers the necessary flexibility for adapting the data distribution but also ensures the necessary performance. Finally, the separation of functionality (Actor) and control (Conductor) allows the computation and control to be parallelised.

2.4 Constructing interactive simulation systems

2.4.1 Composing an ISS

Using the ISS-Conductor methodology, an ISS can then be developed by selecting the proper components and composing run-time interaction constraints and dependencies between them. The customisation of the component activities is achieved through the knowledge base level of MAs. In the knowledge base of MA, the specification of the component functionality, also called the *capability* of the component, is used for qualification checks when the component is to be included in an ISS. The output of a composition, also called a *story*, is a specification of the constraints governing the interaction among the component instances. At run time, each Component instance is

assigned a *role*† with a unique name. MAs orchestrate the system behaviour according to a *story*. The basic development paradigm is shown in Fig. 2.5.

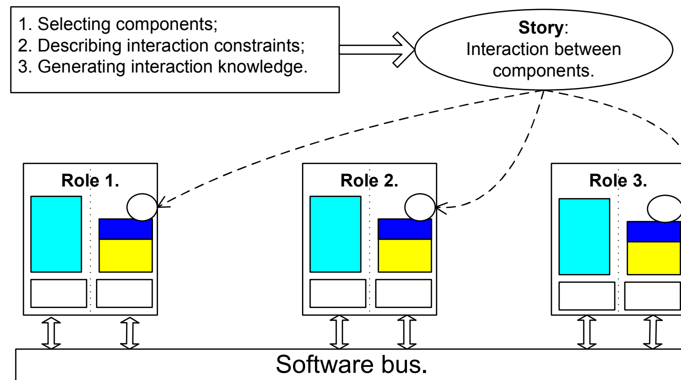


Figure 2.5: A basic paradigm of assembling ISS-Conductor based components.

Using such an approach, the development for an ISS is shifted from realising basic interconnections to describing the high-level interaction constraints. It also introduces two important questions. The first one is how to describe the component capability and the interaction stories and the second one is how to do the qualification check for components. We will leave these questions for the next two chapters.

2.4.2 Run-time framework

The run-time integration between components is through a *software-bus-* like framework, which is normally the infrastructure provided by the middleware that ComAs reside on. The middleware has to provide a number of services demanded by the ComAs: distributed object access, message passing, and data distribution. These services are available in many object oriented middlewares, such as CORBA and HLA. Currently, HLA is used, but the design is not necessarily bound to it. Dependencies on the underlying middleware can be localised in ComAs so that the whole implementation is portable.

Components are directly plugged into the software bus and no intermediate assembly components, like the *containers* in the CORBA Component Model (CCM) [121], are needed. One of the reasons is that ISS-Conductor components are derived from legacy simulation and visualisation systems; a component not only encapsulates the computational routines and data structures of a legacy system but also the necessary logical dependencies between them and the conditions for accessing and invoking them. This integration paradigm shifts the development focus from the assembly of small size computational routines to the specification of constraints on the high-level system behaviour between component instances. These constraints are described as knowledge in the Module Agents.

†In the context of this thesis, a *role* refers to an identifier of a component instance. A *story* is more than a static description of a system behaviour; it refers to the description of interaction constraints.

2.5 Summary

In this chapter we have described the basic architecture of ISS-Conductor. As we have stated in the beginning, the goal of ISS-Conductor is to provide a layered framework for constructing interactive simulation systems, so that the logic control of system behaviour can be separated from the basic coupling details. To approach the goal,

1. ISS-Conductor proposes an autonomous machinery which can wrap the functional units of a legacy simulation or visualisation program, and provide an abstract interface for composing high level interaction among them.

2. It provides an agent framework to encapsulate the computing kernel and the control intelligence of a legacy system, and to incorporate them as a reusable component.

3. At run time, the agents couple the components in a layered scheme: Communication Agents for basic interoperability between components, and Module Agents for controlling system behaviour.

Compared to the solution proposed by Radeski et al., [88] or the one realised in the SIMULTAAN Simulation Architecture (SSA) [91], ISS-Conductor takes a further step: it employs agent technologies to enhance HLA federates; the interconnection interfaces and the interaction control are encapsulated in different agents. The logic structure and the run-time flow of data and activities between component instances are described as knowledge in the agents. No special centralised co-ordinator is employed to control the system behaviour. Using software component and agent technologies in constructing ISSs is a novel approach. Compared with the architecture proposed by the CCA and Bond, ISS-Conductor takes the advantages of available advanced middleware. It views the principal ISS module as components, and focuses on the activity control between them.

Chapter 3

Agent based activity orchestration

In this chapter, we first give a functional description of the architecture and then discuss how the agents control the run-time system behaviour. We focus on a number of innovative designs in ISS-Conductor: capability and story modelling mechanisms, and scenario execution paradigms*.

3.1 An ISS as a multiple Module Agents system

First, we give a short review on the basic concepts introduced in the preceding chapter. An ISS-Conductor component has an *Actor* and a *Conductor*, which respectively encapsulate the functionalities of a simulation or visualisation system and control the run-time activities. Using ISS-Conductor components, an ISS is realised as a collection of component instances with different *roles* and a *story* of the interaction constraints between them. A story is designed based on the *capabilities* of the employed components. The *Module Agents* in the Conductors use the capability and story to control the run-time system behaviour.

As we have discussed in chapter two, an MA achieves its deliberative control on run-time activities using a reasoning kernel, as shown in Fig. 2.4. The *knowledge base* of the reasoning kernel not only contains the descriptions of *story* and *capability*, but also contains the control intelligence for reasoning on activities and for interacting with the other MAs. An MA can thus be described as a machinery which contains four functional components: a *controller*, a *world model*, a *story* and a *capability*, as shown in Fig. 3.1. The *capability* serves as a sort of *expert system* for answering "*what can I actually do?*". The *story* indicates "*what am I expected to do?*". The *world model* shows "*what is going on in the environment?*". The *controller* realises general strategies for managing the *world model* and *capability*, for interpreting the contents of a story and for controlling the Actor.

*This chapter is based on publication: *Z. Zhao, G. D. van Albada and P. M. A. Sloot*. "Interaction Scenario: Orchestrating Agents in a multi-agent System", in the proceedings of the 4th workshop on Agent-Based Simulation, ISBN 3-936-150-25-7, Montpellier, France, 2003. An extended version has also been accepted by internal journal of simulation transaction.

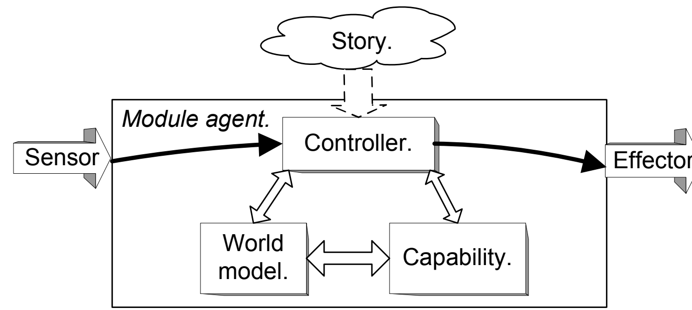


Figure 3.1: A logical view of the functional components in an MA.

The chapter is organised as follows. In 3.2 and 3.3, we describe the basic models of component capabilities and interaction stories. In 3.4 and 3.5 we discuss the design of world model and controller. After that, we discuss three execution paradigms of interaction story in 3.6.

3.2 Inherent functionality: component capability

The functionality of a component defines its capability to serve the others, which determines the spectrum of behaviour that a component can perform.

3.2.1 Basic model

A traditional and also widely used method to describe the behaviour of a system is through sequences of states or actions. A classical model is the Finite State Machine (FSM) model [122]. A typical example is the activity diagram [123], in which the *states* are action or sub-activity states and *transitions* are triggered by completion of the actions or sub-activities in the source states. An activity diagram captures the nature of system behaviour using dependencies between activities and is an important method to model software behaviour. In ISS-Conductor, the component capabilities are modelled using activity diagrams. The dependencies between activities are described using the execution states of activity performance, data, and condition guards. The quality descriptions of the activities are also included as part of the capability for the purpose of component selection and run-time performance adaptation. The component capability is thus defined as 5 elements: (*Actions, States, Data, Transitions, Quality*) where:

1. *Actions* is a set of activities that the component can perform. It always includes an initial action and a set of terminal actions. Each action is associated with two lists of shared data classes for indicating its input and output of data objects respectively. Before an action can be executed, the instances of all the shared classes in its input list have to be available in the input buffer.

2. *States* is a set of states that describe the possible execution status of the actions. It consists of two non-intersecting subsets: $\{S_{unfinished}\}$, for describing the unfinished states, which always contains one initial state and a number of proceeding states, and $\{S_{finished}\}$, for describing the finished states. At run time, the state of an action always starts from the initial state, then shifts to the proceeding states, and finally one of the final states.

3. *Data* is a set of typed data objects which can be either internal or sharable.

4. *Transitions* is a set of transitions between Actions. A transition is described using the state of the starting action, an event and a set of guard expressions. A transition is active when the guard expressions are evaluated to true and the instances of the shared classes described in the input list of the target action are available in the input buffer. An action is called *doable* when it has an active transition from the current action.

5. *Quality* specifies the quality attributes of the component activities and data. In Chapter 5, we will have more discussion on this point.

The capability can also be represented by an activity-transition graph. Assume we have a simulator for solving some equation. It has 5 actions $\{$*Start, InitSimulation, DoStep, ExportResult, Stop*$\}$. The action executions have four possible states: $\{S_{unfinished}\}$=$\{$*toDo, doing*$\}$ and $\{S_{finished}\}$ = $\{$*succeed, failed*$\}$. The initial setting for the computational and control routines is represented in the data object *Setting*; the computational results are formalised as data object *Result*. The activity-transform graph is depicted in Fig. 3.2.

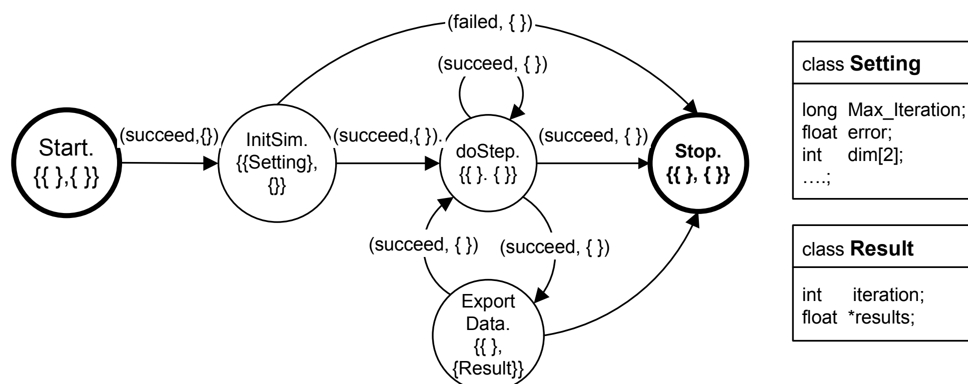


Figure 3.2: A partial activity-transition graph of the example. In the description, actions: Start, DoStep and Stop do not have dependencies on data objects, the action InitSimulation requires a Setting *object as its input, and the action* ExportData *has a* Result *object as its output. In the example, the action* DoStep *has a transition with a condition guard: error > Setting.error, which means the action will only be performed when the error is larger than a given bound.*

The ISS-Conductor is designed on top of object-oriented middleware. The issues related to the ownership of the objects such as object creation, destruction and update

will be handled by the services from the underlying software bus. If an object in the input class list of an action is not updated, the controller will request the owner of the object to update the content before executing the action. If an action has an object in its output list, and the component does not hold its ownership, then the action will issue a request to the software bus to negotiate the ownership before performing the update operations.

3.2.2 Capability modelling for the human interaction involved components

The run-time behaviour of a human-interaction involved component is influenced by both the MA and the human user, thus the human actions have to be considered when modelling its capability. Before discussing the details, we shall first take a look at the basic structure of a stand-alone interactive system.
From the perspective of task processing and human interaction, the functionality of an interactive visualisation system can be described as a hierarchical structure using the notions from Activity Theory [124, 125]†. The top layer is a set of *tasks* supported by the system, the middle layer is a set of goal-directed *subtasks* which can be performed by the user to realise each *task*, and the bottom layer is a set of *operations* for carrying out each *subtask*. An *operation* can be mapped onto an element in the user interface. The interface manager of the interaction system ensures that the dependencies between tasks and interface elements are handled. Fig. 3.3 shows an example of an interactive visualisation tool which allows three basic *tasks*: *selecting a task*, *exploring data* and *stopping the system*.

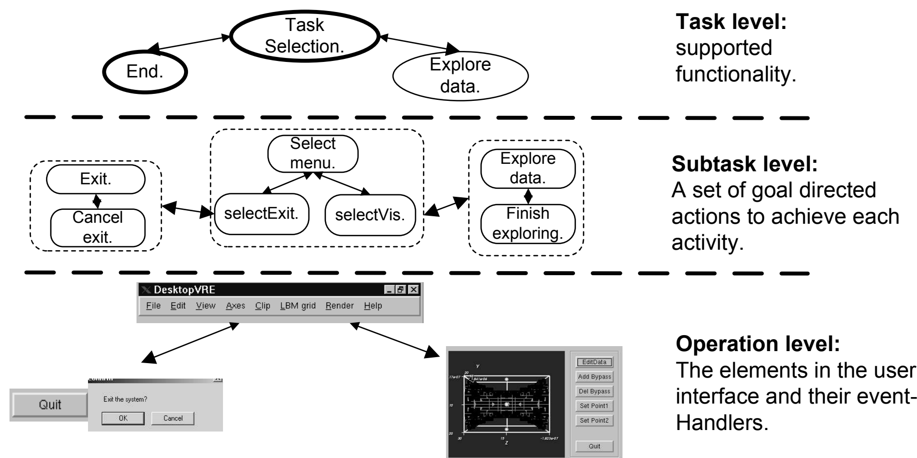


Figure 3.3: A simple model of human interaction involved systems.

The capability model of an interactive system needs to meet two requirements, first, the run-time behaviour of the interactive system can be controllable by the activities in the capability, and second, it inherits the legacy support for human-centred

†To avoid the unnecessary confusions with the capability model, we use the terms *task* and *subtask* instead of *activity* and *action* as the notions.

interaction. In principle, the activities of the system can be modelled at any level of *tasks*, *subtasks* and *operations*. However, it is not preferable to model the component activity at the operation level, because when the legacy controls for the dependencies between interface elements are taken by the Module Agent, the concurrency between the user activities and the agent control tends to introduce a large state space. Currently, we use the task level. Each task is modelled as two actions: *enable* and *disable* which influence the user interaction by enabling and disabling the interface elements respectively.[‡] The capability also includes actions for handling I/O operations. Although not being modelled as the actions in the component capability, the human actions are important in describing the dependencies between the agent actions. A suitable way is to model them as states: the subtasks that the user is performing are modelled as the state of the user behaviour. The other part of the capabilities, such as *Events, Data, Transitions* and *Quality* can be derived in the same way as for a non-interactive component. Fig. 3.4 shows the activity-transition graph of the Fig. 3.3.

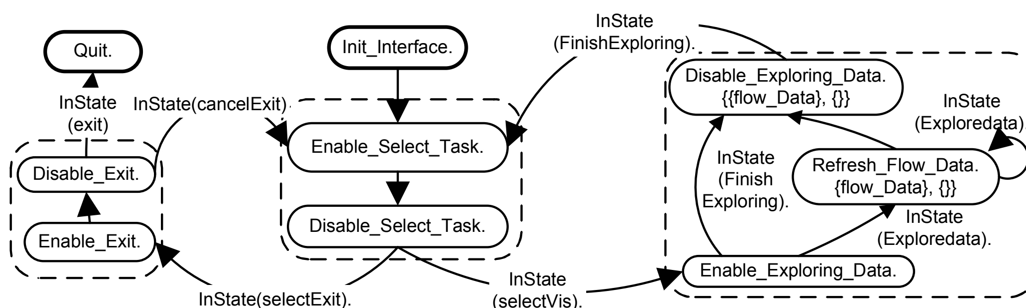


Figure 3.4: Capability modelling of the components involved with human interactions. A partial activity-transition graph of Fig. 3.3. The term InState *describes the state of user activities.*

3.3 Interaction: story and scenarios

A *story* provides rules to steer the run-time behaviour of the component instances. To decrease the complexity, a story is divided into a number of simpler fragments, called *scenarios*. Those *scenarios* can be reused in different *stories*. A scenario can be, in principle, specified in a number of ways, e.g. activity diagrams [126], state charts [127] and Petri Nets [128]. Because of the well-established theoretical framework and more importantly the suitability for representing the common flow patterns and concurrency dependencies [129], a Petri Net based approach [130] is adopted.

[‡]Enabling and disabling interface elements is supported by most of the user interface development toolkits.

3.3.1 Place transition net

The concept of a Petri Net was originally developed by C. A. Petri in the 1960s. It has been a widely applied for modelling system behaviour, in particular concurrent activities. *Petri Net* has actually become a generic word referring to a body of research such as elementary net theories, place transition graphs, and high level graphs [131, 132]. Place transition graphs are a sort of automata that can handle relations between conditions and the occurrence of events [130].
A place transition (PT) graph can be specified as a triple, (*SP, ST, SF*) [133] where:

1. *SP* is an finite set of *places*;
2. *ST* is an finite set of *transitions*. $SP \cap ST = \emptyset$;
3. *SF* is a map between SP and ST, and between ST and SP. $SF \subseteq (SP \times ST) \cup (ST \times SP)$.

A number of concepts are used to specify the properties of PT nets:

1. *Weight* is a map between SF and natural numbers $\{1,2, ...\}$. In this chapter, we only use the nets that have equal weights, 1, for all links in the SF.
2. A *Marking* of a PT graph is a map between SP and $\{0, 1, 2, ...\}$. M(p) denotes the number that marks the place *p*, it also reflects the number of *tokens* in that place.
3. *Pre-set of x*, denoted as $\bullet x$, is defined as $\bullet x = \{ y \in SP \cup ST \mid (y, x) \in SF \}$, $x \in SP \cup ST$.
4. *Post-set of x*, denoted as $x\bullet$, is defined as $x\bullet = \{ y \in SP \cup ST \mid (x, y) \in SF \}$, $x \in SP \cup ST$.
5. *A transition T is enabled* when each element of its pre-set has at least one token. $\forall p \in \bullet T, M(p)>0$.
6. A enabled transition *A* can be *executed*, also called *fired*. The execution will update the marking of the net. The update rule is that all elements in the pre-set of the transition decrease by one token, and all elements in the post-set of the transition increase by one token.
7. An execution of a PT graph can be described using the sequence of occurred markings. The set of all possible sequences yields a connected graph, named the *marking graph* of the PT graph.

Fig. 3.5 shows a model of data production-consumption relation. On the left side, a PT graph contains five places: *a, b, c, d* and *e*, and four transitions: t1, t2, t3 and t4. The initial mark is (1, 0, 0, 1, 1, 0) and t1 is the first enabled transition. After executing t1, the place *a* passes one token to *b*; the mark of the net is updated as (0, 1, 0, 1, 1, 0) and t2 is another enabled transition. The marking graph shown on the right side of the figure describes all the possible marks.

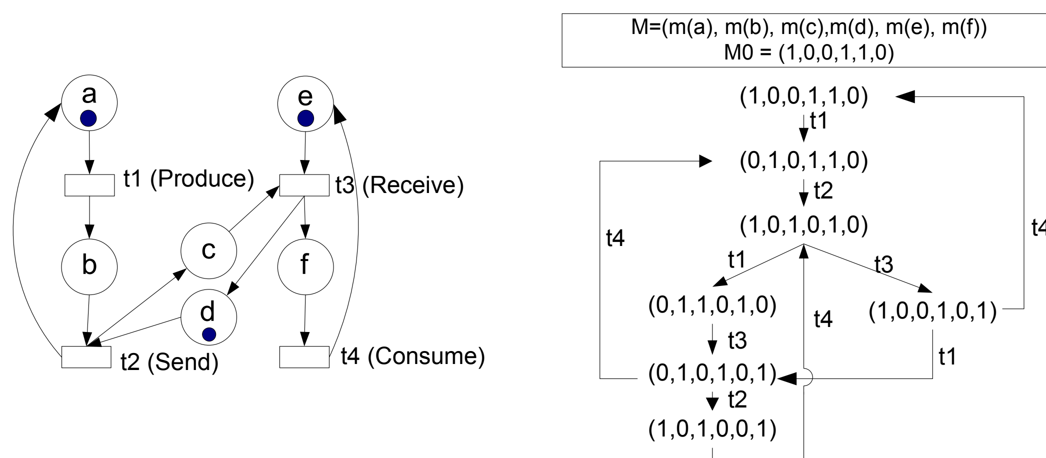


Figure 3.5: A PT net based model of data production-consumption relation.

3.3.2 Scenario representation

A simple PT graph captures the *qualitative* properties between the concurrent activities, but not the *quantitative* properties of the conditions for a specific transition. A common solution is to associate additional information with either the nodes (places and transitions) or the relation links. In ISS-Conductor, the extension is added to places and the links between places and their post sets.

In a scenario, the activity dependencies between the roles are modelled using an extended PT graph, named a *scenario net*. It models the interactions using two types of dependencies between the activities of different component instances. The concurrency dependencies between them constitute the first type, which are represented as the relation links between places and transitions, and the tokens of the places. The second type of the dependencies are the specific conditions for each action, which are represented as the control expressions in the pre-set of the transition and in the links between the transition and its pre-set.

In a scenario net, transitions and places have unique names. Transitions are used to specify activities or nested scenario nets. When specifying an activity, a transition contains an action and a role name, where the role is expected to perform the action at run time. The role is called the *responsible* role of the transition. When specifying a nested scenario net, a transition contains the name of a scenario net and a special action called *Do_Scenario*. Places and the links between places and their post sets are used to describe the conditions. A place is optionally associated with a set of expressions, named *place expressions*, which contain three subsets, for describing the initial conditions, control conditions and the state-modification rules of the place. Each link between a place and its post set is optionally associated a set of guard expressions. We will discuss the semantics of these expressions later. In the expressions, parameters are accessible by all roles when they are updated by the place expressions, or only accessible by a specific role when they are updated by the world model of that role. We use PR({P}) to represent all the parameters that are read by the place expressions

of the place set {P} or the guard expressions in the links between all the elements in {P} and their post sets, and use PW({P}) to represent all the parameters that will be updated by the place expressions of the place set {P}.
The execution of a scenario net is dependent on its marking, the place expressions and the guard expressions in the links:

1. The initialisation of a scenario net takes two steps: it first assigns the initial marking M_0 to the places, and then executes the initial expressions in all places of the scenario net once.

2. *A transition is enabled* when all the places of its pre-set have at least one token, the control expressions in all these places are evaluated as *true*, and the guard expressions in all the links between the transition and its pre set are evaluated as *true*.

3. If the action in an enabled transition is doable, the transition can be *executed* (also called *fired*). The execution will update the marking of the scenario net. It first executes all the state-modification expressions of the places in the pre-set of the transition, and then updates the tokens of all the places in the pre-set and the post-set of the transition as in a normal PT graph.

The execution of a transition updates the state of the scenario net. The basic rules of the PT graph handle the concurrency dependencies between the activities by changing the marking of the net, and the execution rules for the place expressions update the control conditions for the activities. Since the responsible role of a transition will evaluate all the expressions in the places of its pre-set and in the relation links between them, it needs to have right to access the parameters in the expressions.

3.3.3 Transitions and actions

In a story four special actions, named *story-control actions*, are also defined:
Start_Scenario, *End_Scenario*, *Synchronisation_Actions* and *Do_Scenario*.

1. In a scenario net, one and only one transition contains *Start_Scenario*. The transition defines the synchronisation point for starting the scenario net. The role that is responsible for doing the *Start_Scenario* action is the *responsible role* for the scenario net.

2. In a scenario net, one and only one transition contains *End_Scenario*. The transition defines the synchronisation point for stopping the scenario execution.

3. A scenario net may contain a number of *Synchronisation_Actions*. The transitions define points for one or more roles to synchronise their activities.

4. A scenario net may contain a number of *Do_Scenario* actions. As we have mentioned, a *Do_Scenario* action defines the entrance to a nested scenario net. The responsible role of the nested scenario net is responsible for doing the *Do_Scenario* action.

The story-control actions are always doable. The actions specified in the scenario net must either be defined in the capability of a role or be story-control actions. In a scenario net, only the key activities of the involved roles need to be described, and the capabilities of the roles are responsible for searching intermediate actions to link them.
Let's take an example. There are two components a *Producer* and a *Consumer*, which capabilities are shown in Fig. 3.6. We want to build a simple scenario for three roles: *Producer_A* is an instance of the component *Producer*, and *Consumer_A* and *Consumer_B* are two instances of the component *Consumer*. In the scenario, *Producer_A* produces data for both *Consumer_A* and *Consumer_B* 10 times. Data transmissions between *Producer_A, Consumer_A* and *Consumer_B* are through a software bus and use a publication/subscription mechanism, which means each data object produced by *Producer_A* can be consumed by both *Consumer_A* and *Consumer_B*. The *Producer_A* only continues when the *Consumer_A* and *Consumer_B* both finish their consumption (controlled by the *Synchronisation_Action* at transition Sat5). In the scenario net, the place-expressions are only specified when they are not empty. The responsible role is *Producer_A*. At its run-time, the parameters SA and SB increase by one after the *Consumer_A* and *Consumer_B* consumed a data. The expressions in Sap2 and Sap7 control the branch after the Sat5.

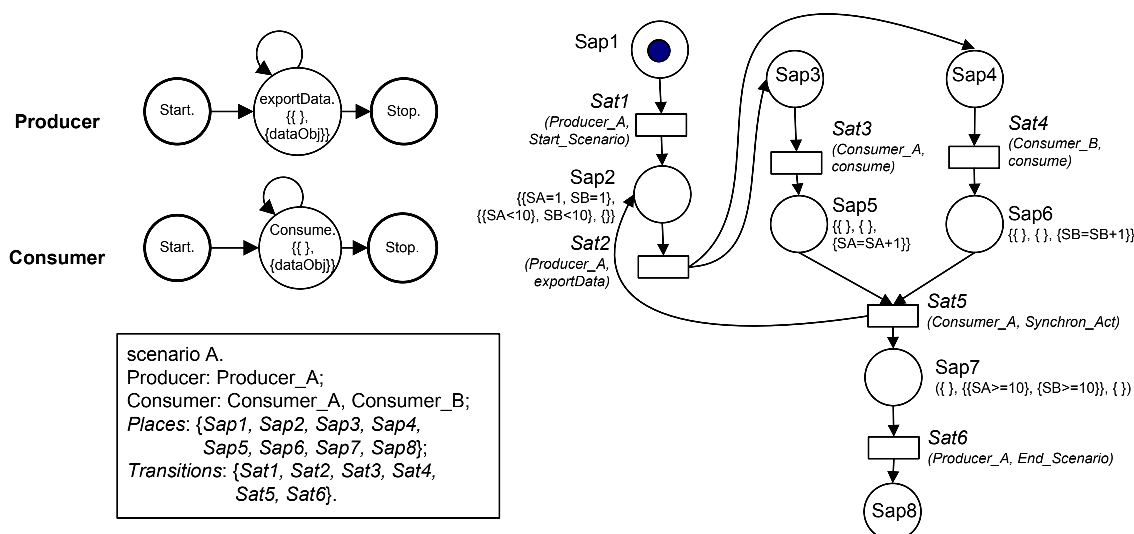


Figure 3.6: A sample scenario for roles Producer_A, Consumer_A and Consumer_B. More examples will be discussed in the next chapter.

3.3.4 Story: a scenario-net instance

A story is a scenario-net instance. It may contain a number of nested scenario nets which are also called scenarios of the story. When the *End_Scenario* action of a story has been executed, the system will exit. The responsible role for doing the *End_Scenario* action broadcasts an exit message to all the roles in the system, and

the peer roles will do the actions that are in a path leading to one of the *terminal actions* in their capabilities.

3.4 World model

To behave rationally in a story, a role has to know not only its own execution status but also the progress of the other roles. Each role has to track the state of the entire system in order to make correct decisions on its activities. The *world model* provides the necessary services.

3.4.1 Basic structure

The world model tracks and processes the changes of the external world using a uniform structure $\{(parameter, observations, perception)\}$. *Parameters* are the things that are being tracked and *observations* are the temporary value of parameters, which are ordered by their time stamps. The perception analyses the observations, both the value and their time stamps, and maps them to a set of qualitative descriptions or obtains the latest value, called *belief* of the parameter.
Based on the type of information, parameters are classified in five groups:

1. *Agent world* related parameters are the names of the involved roles in the story. The perceive function returns the believed states for the role. The state of a role is determined based on two issues: if the role is present in the system, and if the role has recently updated its state. Four states are defined: *never heard of, is updated, is not updated* and *has disappeared*. The semantics and the transitions between them will be discussed later.

2. *Story* related parameters are for scenario changes. The perception function returns its believed story state, which includes the current scenario net.

3. *Scenario* related parameters are for the marking of the scenario net. The perceive function returns the believed marking for the scenario.

4. *Execution* related parameters are for activities and their states. The perceive function returns the believed current action and its state.

5. *Data* related parameters are the names of data objects. The perceive function returns the believed value of object attributes.

3.4.2 Perception and uncertain belief of the agent world

For the agent world, *perception* can use the value of the observations as well as statistical functions, e.g. minimum, maximum and average of the intervals between the time stamps of the observations, and a *belief-transition* graph to derive the belief of a *parameter*. A belief-transition graph is a state machine based model, in which

the states describe the possible belief and the guards in the state transitions are described using the statistical functions or the values of the observation. Initially a role perceives all the peer roles as *never heard of*; after receiving state-update messages, the beliefs of the corresponding peer roles will be turned into *is updated* or *is not updated* depending on the time intervals for updating new states; finally, if the role has not received any messages from a peer role for a relatively long interval, the peer role will be perceived as *has disappeared*.

In the *belief-transition* graph, not all the transitions can easily be represented using a single function, e.g. an agent can not distinguish whether a peer role is *not updated* or *has disappeared* when no messages have been received from that role for a period of time. Based on the fuzzy state machines discussed in [134, 135], uncertain belief is introduced to reason on those situations, as shown in Fig. 3.7. Two types of belief are defined: *certain* or *uncertain*. The *uncertain* beliefs are associated with a set of operations, called *proof actions*, which can be invoked for gathering additional information to make the belief certain. The degree of the *uncertainty* is represented by a real number which is between 0 and 1. If a belief is certain, its degree is always 1. When it is a *uncertain* belief, the invocation of the proof actions will change the degree of the *uncertainty*. When the degree of the uncertainty achieves zero or one, the belief will be transferred to a *certain* one. The world model can hold an incorrect belief about the neighbours, e.g. when the network connection temporarily breaks, the neighbours will be perceived as *has disappeared*, but after the connection resumes, the belief will be turned into *is updated*, as shown in the graph.

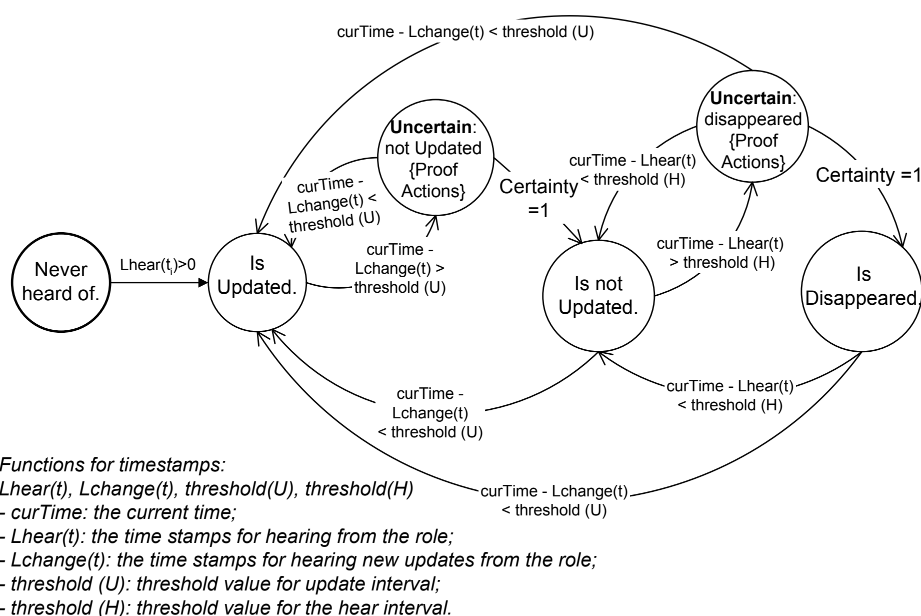


Figure 3.7: A belief-transition graph for deriving the states of neighbour roles.

3.5 Controller

The controller co-ordinates the functional components in an MA and collaborates with the other roles to carry out the story execution. In more detail, the controller processes the information observed by the sensor, updates the world model, finds suitable actions from the story and capability, and controls the execution of the actions.

3.5.1 Collecting observations

The first thing that the controller does is to collect the information observed by the sensors. The sensors are actually the ComA of the Conductor. There are basically three types of information which could be observed by the ComA. The first one is signals that the software bus passes to the ComAs, which are normally generated by the protocols of the underlying middleware. The second one is messages that components send to each other. The purpose of a message is either to update or to query the state information. The difference between them is that the second type of messages expects a reaction from the receiver. The third one is the reflection of the new value of data objects and their attributes.
The ComA in the Conductor observes the events from the external world and passes them to the controller in the MA. The controller then generates more specific events for the world model to update corresponding parameters. The controller checks regularly whether the world model has any proof actions that need to be invoked.

3.5.2 Action execution control

The controller also controls the action execution. In general, the controller handles two types of actions: those specified in the story or the capability, and those that are a response to the normal events, including *proof actions*. The first type of actions can only be executed when the previous one has been finished, while the second one can be executed at any time. For the first type of actions, if it is not a story control action, it will have to be sent to the Actor. The controller handles the protocols for action sending and searching using the states provided by the world model, as shown in Fig. 3.8.

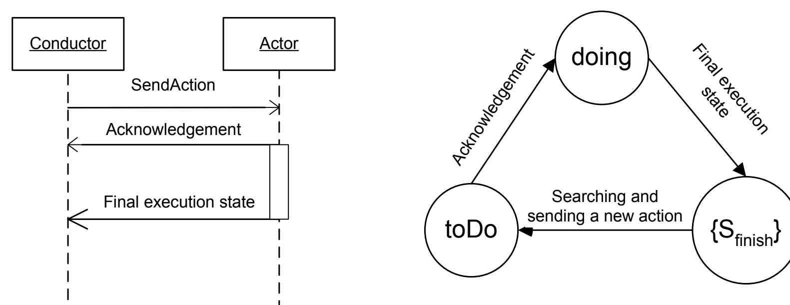


Figure 3.8: The action control between an Actor and a conductor, and its reflection in the world model.

The action requested by the story is possibly not immediately doable by the capability, but can be doable after a certain number of intermediate actions. For this kind of actions, the controller considers the states of the action that is executed by the Actor as a nested state, and uses it to control the update of the story state, as shown in Fig. 3.9.

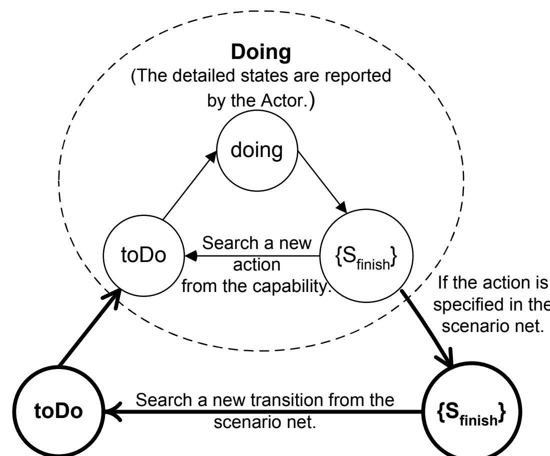


Figure 3.9: The execution states of the actions.

Another main function of the controller is to collaborate with the other roles to execute a story. This will be discussed in the next section.

3.6 Story execution

At run time, the MA in a component interprets the story and controls its behaviour. The MAs collaboratively orchestrate the overall system behaviour in three possible paradigms: distributed, hierarchical and centralised.

3.6.1 Basic paradigm: distributed scenario execution

In *distributed scenario execution* each role maintains its own execution state of the story, and independently finds enabled transitions from the scenario net using its local states. Execution proceeds through four basic phases: finding actions, executing actions, updating the local story state and synchronising the state with the other roles.

Finding actions

Searching for an action normally takes two steps. The MA first finds an enabled transition from the scenario net, and then checks if the action defined in the transition is doable. A story-control action is always doable and a normal action has to be approved by the capability. When the action in the enabled transition is not directly

doable, the searching rules will find an intermediate action from the capability, which leads a path to it.

Action execution and concurrency control

Story-control actions are executed by the Conductors, and the normal actions are executed by the Actors. Before an action is executed, its safety with regard to possible concurrently executed actions has to be checked. A concurrency conflict occurs when there are two transitions for which two different roles are responsible, and the execution of one transition might disable the condition of the other one. Fig. 3.10 shows an example of concurrency conflicts and the possible illegal markings. The scenario fragment contains four transitions: role A is responsible for T1 and T3, and role B is responsible for T2 and T4. When executing in the distributed paradigm, both role A and B have an initial mark of the scenario net: (1, 0, 0, 0), and both of them have an enabled transition: T1 and T2 respectively. If both of them simultaneously execute the transitions, the mark of the scenario net will be turned into (0, 1, 1, 0), which is apparently invalid in the actual marking graph of the scenario net, as shown on the left bottom of the figure. In the scenario net, T1 and T2 have concurrency dependencies, and are called **critical transitions**.

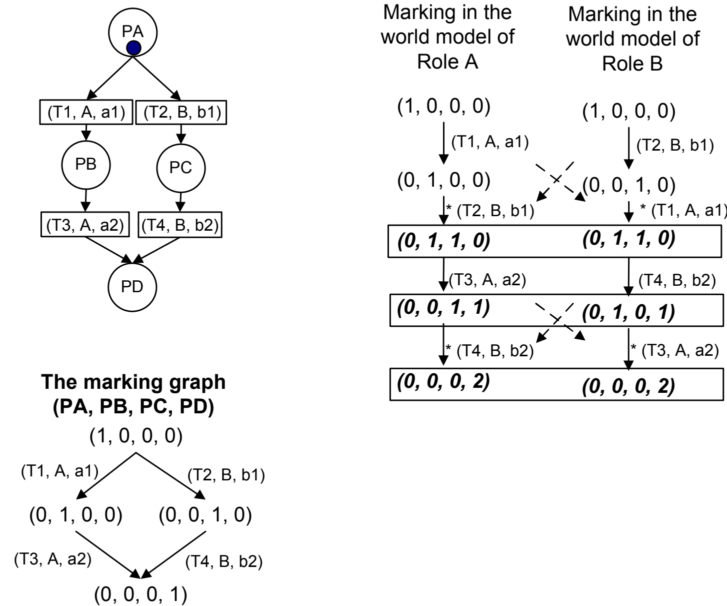


Figure 3.10: A scenario fragment and its marking graph are shown on the left side. The right side shows a possible execution sequence of role A and B, when they do not apply any concurrency controls. The dashed arrows indicate the marking changes that are perceived by the peer role; the markings in Italic font are invalid.

The concurrency conflicts can be checked using the following rules. Assume that role A and B are responsible for two different transitions T_i and T_j in a scenario S. The transition T_i and T_j are critical transitions when executing either of them might

change the condition of the other: the number of tokens in the pre-set or the condition expressions in the places and in the relation links. In more detail, the critical transitions can be identified using $\bullet T_i \cap \bullet T_j \neq \emptyset$ or $\mathrm{PW}(\bullet T_i) \cap \mathrm{PW}(\bullet T_j) \neq \emptyset$ or $\mathrm{PR}(\bullet T_i) \cap \mathrm{PW}(\bullet T_j) \neq \emptyset$ or $\mathrm{PW}(\bullet T_i) \cap \mathrm{PR}(\bullet T_j) \neq \emptyset$. Role R is an involved role for a critical transition T_i when R is responsible for T_i, or it is responsible for a transition T_j which is in *conflict* with T_i.

To execute a critical transition, a role has to negotiate with the other involved roles to ensure mutual agreement. A negotiation is designed based on the algorithm described in [136]. The local time of the roles is used for the comparison; the one with the smallest value wins.

State update and synchronisation

After a transition has been executed, a role first updates its local state of the scenario net and then causes the other roles to synchronise their local states. A role broadcasts an update-request message to the other roles when it has updated its local state. The peer roles synchronise their local states by mimicking the execution of the state after receiving the state update request message. The synchronisation operation can only take place once for each request.

The services provided by the Run Time Infrastructure of HLA, e.g. ordered-message delivery, can not easily handle the synchronisation, because message delivery is not always reliable, e.g. due to a temporary loss of the connections between roles. Processing the messages in a Receive-Ordered (RO) way, a role cannot ensure that all its requests for state-update have been received by the peer roles, neither can it ensure that it has received all the requests from the peer roles. On the other hand, with Time-Stamp-Ordered (TSO) message delivery, the grant of the federation time will be blocked by one agent when it is out of function. A high-level control for the synchronisation is needed.

To record the executions of transitions, each role maintains two groups of data structures in its world model. The first one is called *master table*, which records all the transitions that it has executed, and each item in the master table is also associated with an *acknowledgement table*. And the second one is called *slave table* which records all the transitions for which update request messages were received; a separate *slave table* is maintained for each peer role. Fig. 3.11 shows the basic structure.

These tables ensure that any executions performed by a role will be synchronised by all the peer roles exactly once. First, the sequence of the transitions executed by a role is tracked using the *master table*, the acknowledgement tables check if the transitions have been reflected by all the peer roles. Second, the *slave* tables record the histories of the transitions executed by the peer roles, which guarantees that the local update for each transition only takes place once.

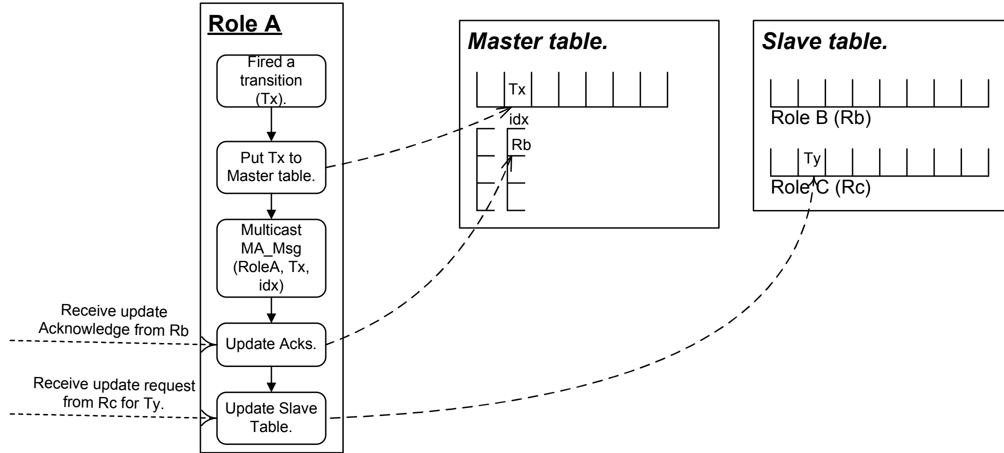


Figure 3.11: Data structure for state synchronisation.

3.6.2 Hierarchical execution paradigm

In the distributed paradigm, the state update messages are broadcasted. If the number of roles or transitions increases, the number of messages will also dramatically increase, and the massive number of small size messages will degrade the system performance. The hierarchical execution paradigm is proposed to overcome this problem. It intends to limit the number of messages by only sending the update requests to the roles that really need them. First we will define some basic concepts.
We define that two roles R_a and R_b are **tightly dependent**, denoted as *tDep*(R_a, R_b), when T_a and T_b which are the responsibility of role R_a and R_b respectively satisfy at least one of the following conditions: $(\bullet T_a \cap T_b\bullet) \cup (\bullet T_b \cap T_a\bullet) \cup (\bullet T_a \cap \bullet T_b) \neq \emptyset$ or $(\mathrm{PW}(\bullet T_a) \cap \mathrm{PW}(\bullet T_b)) \cup (\mathrm{PW}(\bullet T_a) \cap \mathrm{PR}(\bullet T_b)) \cup (\mathrm{PR}(\bullet T_a) \cap \mathrm{PW}(\bullet T_b) \neq \emptyset$. Two roles R_a and R_b are **loosely dependent**, denoted as *lDep*(R_a, R_b), when T_a and T_b are not tightly dependent, but there exists a sequence of roles $\{R_1, R_2, \cdots, R_n\}$ which has *tDep*(R_a, R_1), *tDep*(R_1, R_2), $\cdots$, *tDep*(R_n, R_b). And two roles are **dependent** when they are either tightly or loosely dependent. From the definition, we can see if a scenario net is connected, which means no transitions are isolated from the others, any two roles are dependent.
The basic idea of the hierarchical paradigm is that all the roles which are tightly dependent should receive the update request messages from each other. Therefore, the distribution of the update-request messages will be multicast instead of broadcast. To construct the multicast groups, a **dependency tree** is proposed:

1. The nodes in the tree contain a non-empty set of roles, the root node only contains one role that is the responsible role for the scenario. A role can only belong to one node in the tree. The {R} is the set of all the roles in the node.

2. For a role R, all the roles that are tightly dependent with it must belong to one of the possible nodes: the same node with R, its parent node, or one of its child nodes.

3. Any two roles R_a and R_b which belong to the same node must be either tightly dependent or loosely dependent. And if they are loosely dependent, there must exist a set of roles $\{R_s\} = \{R_{s1}, R_{s2}... R_{sn}\}$ which has $\{R_s\} \subseteq \{\mathrm{R}\}$ and tDep(R_a, R_{s1}), tDep(R_{s1}, R_{s2}), tDep(R_{s2}, R_{s3}), $\cdots$, tDep(R_{sn}, R_b).

4. Any two roles R_a and R_b which neither belong to the same node, nor to parent-child node pair must not be tightly dependent.

In a connected scenario net, any two roles are dependent, which means a dependency tree can always be derived. Using the dependency tree the multicast groups for distributing state-update messages can then be allocated. All the roles that are in the same node or the parent-child nodes will be in one group, and the roles that do not belong to the same node or parent-child nodes will be in different group. The multicast groups are handled using the data distribution services provided by the underlying middleware, which will be discussed in the next chapter.

3.6.3 Centralised coordinator paradigm

The final execution paradigm is to interpret a scenario using a centralised co-ordinator. An important reason for employing this paradigm is that the cost of the negotiation operations, especially when the number of critical transitions and the number of their involved roles are large. The centralised co-ordinator paradigm works as follows:

1. The responsible role of the scenario is set as the co-ordinator.

2. Only the co-ordinator maintains the state of the scenario net, and only the co-ordinator searches the enabled transitions.

3. The co-ordinator searches enabled transitions not only for itself but also for all the other roles in the scenario.

4. When a subordinate role receives a transition sent by the co-ordinator, it will check the doable actions using its own capability. When the action in the transition has been executed, it sends back the execution state to the co-ordinator, and requests for the next enabled transition.

5. Between subordinate roles, no update request messages are sent.

To allow the co-ordinator to check the enabled transition for different roles, the parameters defined in all place expressions should be accessible by the co-ordinator. Therefore, for this paradigm, the subordinate roles necessarily send the value of their private parameters to the co-ordinator during the scenario execution.

3.6.4 Scenario switch and execution paradigm selection

When entering an ISS, a role initialises the story as the first scenario. A new scenario is switched on when a *Do_Scenario* action is executed. The role that executes the *Do_Scenario* action first broadcasts a scenario-switch message to all the roles, then saves the execution state of the current scenario, which includes the mark of the current scenario net and the setting of its execution paradigm, after that it initialises the new scenario. When a peer role receives a switch announcement, it saves the state of the current scenario, and then initialises the new scenario. Currently, no parallel scenarios are allowed, which means at one time, there can only be one active scenario in the system.

When a scenario is started, its responsible role makes a decision on the execution paradigm, which is based on two basic facts: the total number of the involved roles and the total number of critical transitions in the scenario. The facts for designing the rule are that the centralised paradigm can handle the critical transitions more efficiently than the other two paradigms, but its performance will decrease when the total number of the roles is large. The threshold values for the number of roles and the number of transitions are empirically set in the knowledge base. The responsible role announces the decision on the execution paradigm to the other involved roles before executing the action *Start_Scenario*.

Roles choose multicast groups for receiving messages and data objects when entering a scenario. If a role is not involved in a scenario, it will not join any groups for receiving state-update messages. But the messages for scenario switches are broadcasted to all roles. In the next chapter, we will discuss how the adaptation is realised using the distribution routing spaces.

When a role executes the *End_Scenario* action, the current scenario will exit. If the current scenario is the story, the execution of the system will finish. Otherwise, the role restores the state of the previous scenario, then broadcasts a restore message to all the other roles. Exiting a nested scenario means the completion of a *Do_Scenario* action in the previous scenario, the role therefore also needs to update the states of the previous scenario net and announces a state update message to the other roles. The history of the scenarios is tracked by the *master* and *slave* tables of the story.

3.6.5 Handling run-time exceptions

At run time, the execution of a story can have a number of exceptional situations, e.g. some roles never show up in the scenario, join the scenario while the others have already started the execution, or suddenly disappear from the system. These exceptions can have different reasons such as the temporal loss of the network connection, a temporarily unavailable computational infrastructure, or internal errors in the component, which can occur at any time, especially when the run-time environment contains a large collection of heterogeneous computational resources such as computational Grids. To execute a story robustly, the MA employs a number of strategies to handle those exceptional circumstances.

First of all, a scenario can only be started when the responsible role is present. This implies two things: when a story is started, the responsible role for the story has to be present, and a *Do_Scenario* action can only be executed by the role that is responsible for the nested scenario net. The presence of a role is determined by the world model. Secondly, a role is not allowed to enter into a scenario when the other roles have already started the execution. It means that the normal roles of the scenario have to be present in the scenario before the responsible role starts the execution. This strategy is also related to a third one that the transitions that are the responsibility of absent roles will always be considered as *enabled*. Making this assumption avoids that the execution is blocked by an absent role. During the execution, if a component crashes, the other roles will eventually perceive that role as disappeared from the story. If a role disappeared from the story, the other roles will use the same strategy as for an absent role to handle the rest of the scenario. If the responsible role of the scenario crashes, the roles will switch their scenario to *End_Story* to terminate the system. Finally, if the story contains a *invalid* transition, e.g. the action in the transition is not defined in the capability or will never be doable from the current state, the responsible role of the transition will announce a message to the other role that it quits from the scenario. And the other roles will use the second and third strategies to handle the rest of the scenario.

3.7 Summary

In this chapter, we have discussed the core design of Module Agents and the mechanisms for orchestrating interaction among them. As we have discussed in the previous chapter, ISS-Conductor realises the separation between system functionality and application specific interactions using a layered agent framework, in which CoMAs realise the basic communication details and MAs provide an abstract structure to control the activity. The component functionality is modelled as a finite state machine (*capability*), which can be programmed with the other components using a Petri net based mechanism (*scenario net*). In this chapter, we have not presented the experimental results of the development, but from the discussion, we can enumerate a number of design characteristics.

1. Describing the capabilities of the components and the interaction dependencies separately is essential to support scientists to prototype an ISS from high level.

2. The capability of a component is modelled using a Finite State Machine based model. A uniform mechanism is proposed for describing the capabilities of both normal components and components involved human-interaction. In the model, the human activities are modelled as different states based on the activity theory.

3. The interaction dependencies between components are modelled using a Petri net based mechanism, called scenario net. In a scenario net, component activities are described in transitions, and conditions for the activities are described

using expressions in places and in the relation links. The rich semantics of Petri nets can describe the interaction constraints not only from the perspective of data dependencies, as often used in *scientific workflow* systems, e.g. SciRun, Sculf, and GridAnt [16, 137, 138], but also from the concurrency relations between activities. It achieves a paradigm for rapid prototyping of ISSs.

4. The world model plays an important role in the run-time control of system behaviour. In ISS-Conductor, the world model includes fuzzy states in the perception of the other MAs.

5. Three execution paradigms are proposed in MAs. The execution of ISS-Conductor offers more flexible paradigms than centralised control: distributed and hierarchical ones.

In the coming two chapters, we will first discuss the implementation details of ISS-Conductor and then use a medical application as a test to demonstrate the main features of the architecture.

Chapter 4

Implementation and performance analysis

In this chapter, we discuss the implementation of ISS-Conductor. We start with Communication Agents and Module Agents, and then discuss how they are combined in the Actor and Conductor of an ISS-Conductor component, after that we study a test case to investigate the performance characteristics of the system*.

4.1 Communication agents

A Communication Agent has a *data object manager* for managing the structure and contents of data, a *distribution manager* for sharing the data with other ComAs, and a reflex *task-processing engine* for responding to events received from the external world.

4.1.1 Data object manager

In a ComA, the structure of the data is described as *data classes*, which contain a set of *attributes*, and the contents of the data are managed as instantiations of the *data classes*. A class is called *shared* when its instances can be accessed remotely by other ComAs, otherwise called *internal*. The shared classes that have persistent instances are called *shared object classes*, otherwise they are called *message classes*, and their instances are called *shared objects* and *messages* respectively. A shared object class can be syntactically mapped to classes which are defined in the other ComAs. The *data object manager* manages the lifecycle of data objects and provides name services for them. It also buffers the contents of the shared objects that are updated by remote ComAs before the ComA processes them. Inherently, ISS-Conductor predefines a number of data classes in the ComAs:

*Parts of this chapter have been published in *Z. Zhao, R. G. Belleman, G. D. van Albada and P. M. A. Sloot*. " State Update and Scenario Switch in an Agent Based Solution to Constructing Interactive Simulation Systems", in the proceedings of the Communication Networks and Distributed Systems Modelling and Simulation Conference, San Antonio, US, 2002.

1. *Monitor* is an internal class. Its attributes point to the data structure defined in the Actor for tracking and accessing their run-time values;

2. *Task* is an internal class, which describes the structure of the events;

3. *ComA Message* is a shared class, which describes the structure of the information exchanged between the Actor and the Conductor in a component;

4. *MA Message* is a shared class, which describes the structure of the information exchanged between Conductors;

5. *Control Message* is a shared class, which describes the structure of the information exchanged between all Actors and Conductors.

4.1.2 Distribution manager

Using a software bus to communicate, ComAs exchange both shared objects and messages using a publish/subscribe mechanism. The multicast groups for distribution are handled by a *distribution manager* through routing spaces. The routing spaces are defined as two-dimensional planes, where regions are determined by the co-ordinates of two diagonal points. One shared class can only be associated with one routing region at a specific time. At run time, a ComA can change the policies for distributing a shared object by modifying the region of the associated routing space. The basic rule is that ComAs can only exchange a shared object when they have overlapped regions. To simplify the region adaptation, ComAs provide four routings profiles: *inside a component*, *Actors/Conductor only*, *all components*, and *away from the others*, as shown in Fig. 4.1. Four routing spaces are predefined in ComAs: *componentRouting*, *agentRouting*, *controlRouting* and *objectRouting* which are used for delivering ComA Messages, MA Messages, Control Messages and persistent objects respectively, as shown in Table 4.1.

Table 4.1: Messages and their associated routing spaces.

Messages	Routing spaces	Default profiles
ComA message	*componentRouting*	Inside a component
MA message	*agentRouting*	All Conductors
Control message	*controlRouting*	All Actors and Conductors
Data objects	*objectRouting*	All Actors

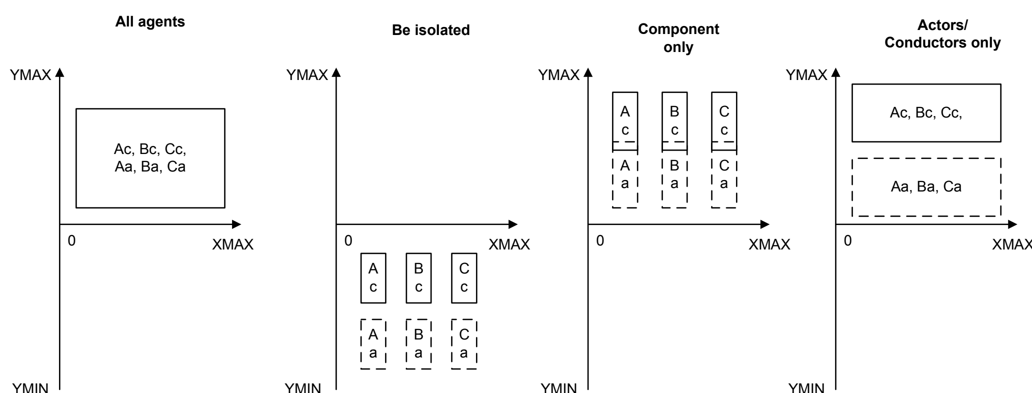


Figure 4.1: Routing spaces and their four default profiles defined for ComAs. Roles are {A, B, C}, the ComAs at Conductor are {A_c, B_c, C_c} and the ComA at Actor are {A_a, B_a, C_a}. The regions for the Conductors use solid lines; and for Actors use dash lines. XMIN, YMIN, XMAX and YMAX are the boundary of the entire region.

4.1.3 Events and action execution

A ComA listens to the software bus and generates tasks for the received events. An event can be a reflection of value changes of a shared object, receiving a message, or a control signal from the software bus. A task has a name, a count for required number of executions, and a content. Tasks are passed to an interpreter via a FIFO queue, and the interpreter searches suitable actions for the task using a lookup table. The interpreter invokes an action and tracks its run-time state using the *Monitor* object.

4.2 Module Agents

In the Conductor, a Module Agent incorporates a *reasoning kernel* to realise the intelligence for controlling the component behaviour. The *reasoning kernel* contains five parts: a *Capability* for describing the basic functionality of the component, a *Story* for describing the interaction constraints with the other components, a *World Model* for tracking the state of the external world, a set of *Control rules* for searching activity and co-ordinating with the other peers, and a *Reasoning engine* for interfacing with the *event interpreter*. The basic structure of a MA has been discussed in the previous chapter.

4.3 Putting it all together

4.3.1 Current implementation

The implementation of HLA specification Version 1.3 and the RTI Next Generation Version 5 is used as the underlying software bus. Using HLA terminology, an ISS is called a *federation*, each ComA represents a *federate*, and the RTI is the *physical*

world for the agents. The sensors and effectors of a ComA are realised by the local RTI library (libRTI). The *data object manager* uses the Object Management services to declare shared classes, to update object values and to reflect the changes of the objects from the RTI. The distribution of data objects is realised using the Data Distribution Management services and time Management services. The run-time system is managed using Federation Management services.
The basic information of a ComA, such as name, type and the location of the capability specification is described in a structure called *ComARole*, which is used to initialise the kernel of an agent. The event-processing loop only starts after the ComA has been initialised as a federate and has joined a federation. Fig. 4.2 shows the basic lifecycle of a ComA. The reasoning kernel of an MA is written in Prolog. In an MA, the task interpreter is implemented using C++; it is coupled with the reasoning kernel using a client-server style. The interface is realised using the Logic Server of Amzi Prolog [139].

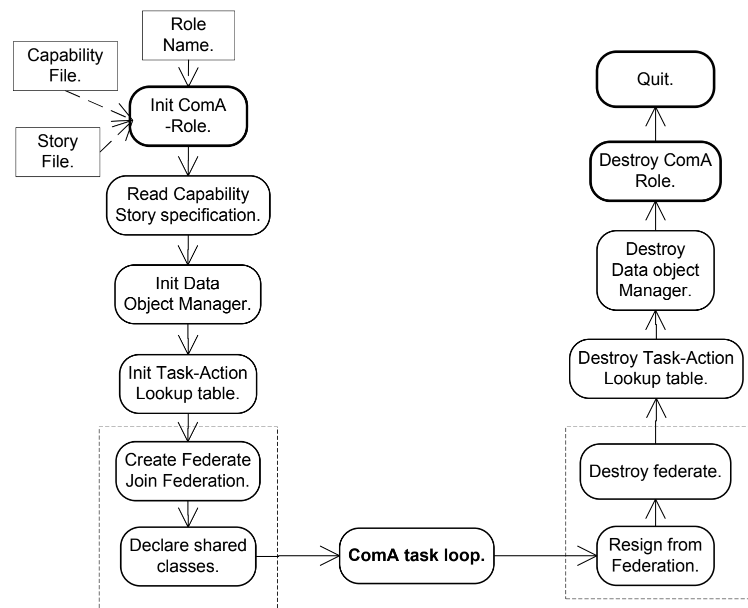


Figure 4.2: A detailed lifecycle of a ComA.

4.3.2 Actor and Conductor

Using the ComA and MA, the Actor and Conductor can be constructed. An Actor wraps the legacy assets of a simulation or an interactive visualisation system. The data structures are encapsulated as internal/shared data classes, and the computational routines are incorporated as actions which are registered in the lookup table of the ComA. Following general engineering principles, the development of an Actor takes a number of steps, e.g. requirement analysis, capability specification, code incorporation, validation, and executable generation. In the next chapter we will discuss these issues with a test case. A Conductor contains a ComA and a MA. The task

processing loop of the ComA and MA are merged, and the lookup table of the ComA only contains the communication-related actions.
The Conductor of a component is equipped with a user interface, which can be launched optionally at run time. The interface presents basic execution information of ComAs, reasoning procedures of MA, data objects' states, the states of the other MAs and a tool for interacting with the other users. Fig. 4.3 shows a snapshot. In the next chapter, we will discuss the utilisation of this tool in more detail.

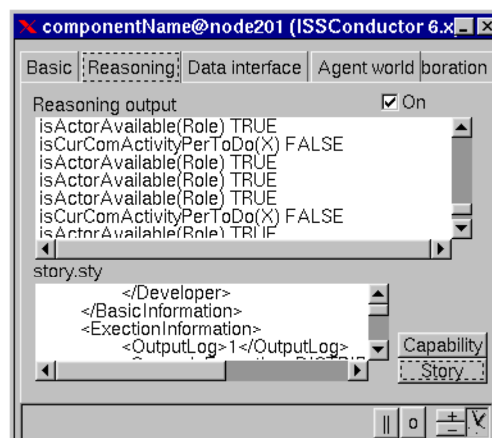


Figure 4.3: The snapshot of the GUI of a Conductor.

4.3.3 Capability and story descriptions

Using the ISS-Conductor architecture, an ISS can be constructed by instantiating suitable components and describing the interaction constraints between the component instances. Each component has an explicit description of its capability. XML schema [110] based templates are defined for describing the capabilities of components, and the interaction story between component instances. The capability template contains sections for describing the data classes, the actions and states, and the activity-transition graph. It is derived from the Object Model Templates (OMT) provided by HLA for documenting the object models for simulation and federation. Using the specification, a tool called *isscTempG* is provided to generate the source framework of the component. The story template allows the user to describe the participating component instances, the projections between their shared classes, and the activity flows.

4.3.4 Run-time configuration files

The design of the ISS-Conductor treats HLA and the Amzi Logic Server as black boxes, their kernels are not modified in the ComA and MA. The run-time configuration files required by HLA and the Logic Server e.g. the specification file for the

federation object and the configuration file for the RTI, and the configuration file for the Amzi logic server, are generated automatically from the specification files of capability and story, as shown in Fig. 4.4.

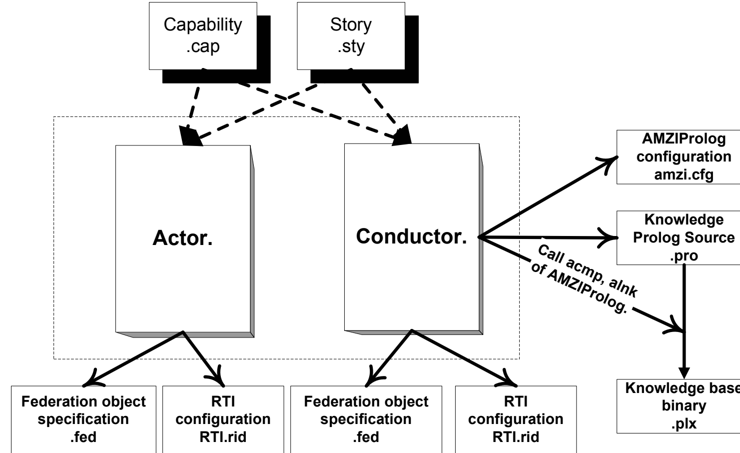


Figure 4.4: Generating run-time configuration files.

4.4 Performance analysis

The performance study of the current implementation focuses on communication related issues, such as latency and throughput for ComAs to remotely update shared objects, and the reasoning related issues, such as overhead for the Module Agent to take decisions on activity control. We use these experiments to briefly evaluate the implementation.

4.4.1 Example components and the test bed

We construct two example components using the current implementation of ISS-Conductor. A *Producer* component (*Producer* for short) maintains a shared object, *dataObj*, which has an adjustable-size attribute called *dataField*. The *Producer* has four actions: *init* for resetting the size of *dataField*, *adaptSize* for increasing the size of *dataField*, *exportData* for updating the value of the object attributes, and *stop* for exiting the system. Each invocation of *adaptSize* doubles the size of *dataField*. A *Consumer* component (*Consumer* for short) contains three actions: *init*, the initial action, *consumeData* for receiving data objects, and *stop* for exiting the system. Fig. 4.5 shows the capability of these two components. Using these two components, we build a simple communication scenario for updating shared data objects.
The test bed is the super computer of the Dutch ASCI research school (DASII) [140], which contains 200 Pentium III nodes, distributed in 5 clusters. The clusters are connected via SurfNet [141]. Each node has one Myrinet card [142] and one fast Ethernet

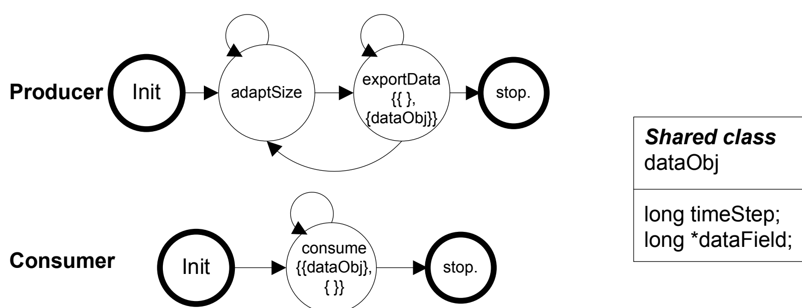


Figure 4.5: Two components constructed for benchmarking ISS-Conductor.

card. The operating system is Redhat Linux. The data transmission between ComAs is reliable.

4.4.2 Delay for remote updating shared objects

The first thing we studied is the delay of updating shared objects. The services for updating shared data objects and for reflecting their changes are provided by the Object Management services from the RTI, which are realised using TAO, a real-time implementation of CORBA [121]. At its basis, the connections use TCP sockets over a fast Ethernet.

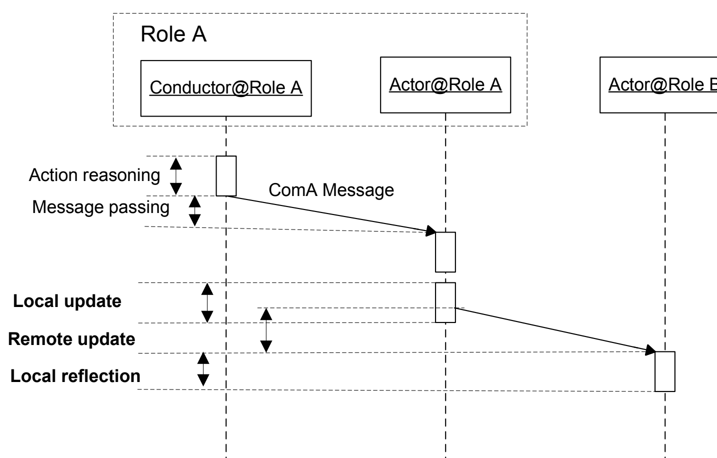


Figure 4.6: Action reasoning and update of shared objects in between run-time roles.

Fig. 4.6 shows three phases for updating a shared object, first the ComA updates the object contents locally, then pushes the changes of the attribute value to remote ComAs which have subscribed to the data class, finally the remote ComAs reflect the data from the local RTI library. The delays of the local-update and the reflection include the overhead of the data object manager in the ComA, and the encoding and decoding of the RTI services. The delays of the remote update include the overhead of the RTI Object Management services, TAO and the TCP sockets. We measure

these delays at the ComA layer and treat them as one. For comparison purpose, the communication performance of a pure TCP sockets has also been measured.
In the experiment, the RTI execution is launched in the fileserver of the cluster das2.nikhef.nl†, and two nodes (node201 and node202) in the cluster are used for executing Consumer and Producer respectively. Each measurement takes 23 time steps, which starts from the attribute size of 16 bytes and doubles the size after each step until 64 Mega bytes. In total, the measurement has been done 50 times.
From the measurements, we observed a number of things. First, the delay for a remote update is larger than the local updating and reflection, as shown in Fig. 4.7. Second, the RTI introduces certain overhead on the communication; for small size objects (smaller than 8K bytes) the delay remains nearly constant as 0.002, which is larger than transferring data using TCP Sockets, as shown in 4.8. Third, the delay for remotely updating linearly increases when the size of the data object increases. In the figures, we observed strange curves in the measurements, for both the RTI and TCP sockets, which are inherent to the Ethernet cards on the DAS II system‡. Improvements are expected when the system is upgraded [143]. Fourth, for large objects, the throughput of the remote update can achieve 8 Mbytes per second, which is comparable to the throughput of TCP sockets.

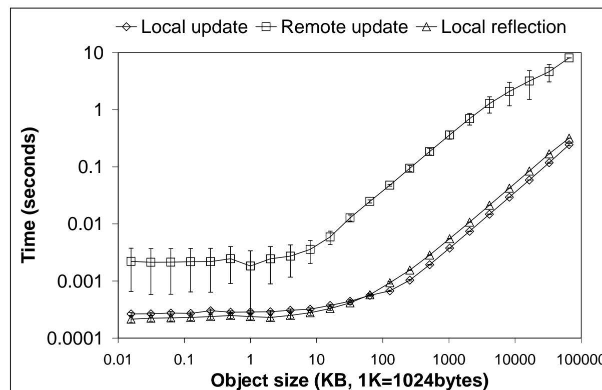


Figure 4.7: The delays of the local update and reflection and for the remote update of a shared object. The error bars indicate the standard deviation at each step.

4.4.3 Location of the RTI execution

In distributed problem solving environments, the RTI execution is a software resource, which can be shared by different applications. How the location of the RTI execution influences the remote update of shared objects is an important issue for executing an ISS. In this experiment, we measured the remote update between two

†The cluster of das2.nikhef.nl is located at University of Amsterdam; fs2, and node201 and node202 are in this cluster.

‡We did not observe the similar behaviour from the measurements between two local Linux workstations. The discussion on this particular issue is out of the scope of this thesis.

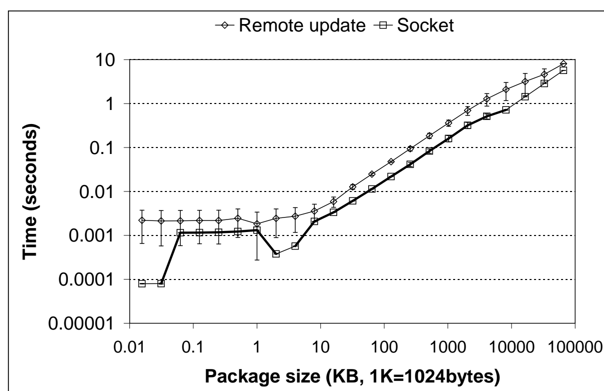


Figure 4.8: Remote update of shared objects and the throughput of pure TCP sockets.

nodes as in the previous experiment, but run the RTI execution in three different locations, as shown in Fig. 4.9.
From the measurements, we find that the delays for the remote update are reasonably close to the original system, as shown in Fig. 4.10. It shows that the location of RTI execution does not have a clear influence on the remote update of shared objects.

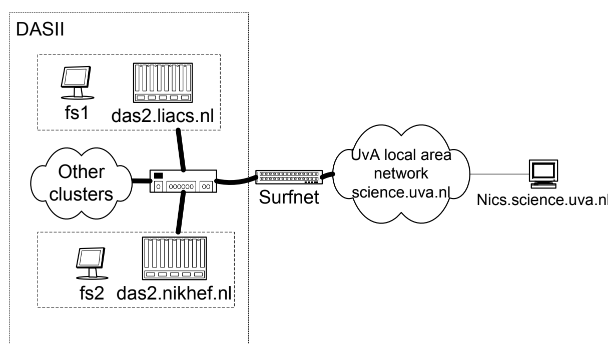


Figure 4.9: The basic architecture of DASII and the configurations of the experiment. The RTI is executed in fs1, fs2, and nics.

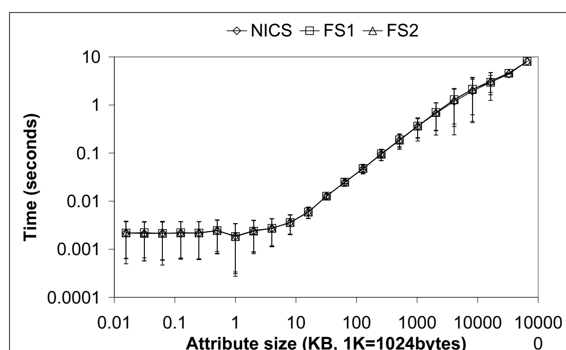


Figure 4.10: Update delay with different RTI locations. The error bars indicate the standard deviations.

4.4.4 Remotely updating objects to multiple Consumers

In an ISS, a shared object is often updated and consumed by more than one consumer, the correlation between the update delay and the number of consumers is important for analysing the overall performance of the ISS. We run the test case in four configurations, one producer with one, two, four, and eight consumers respectively. The experiments are executed in a single cluster; the delays of the remote update between the producer and each consumer are measured. When a configuration contains more than one consumer, the delay T_R, which is defined as the range from the moment that the first consumer starts the reflection until the moment that the last consumer finishes, is also measured.
We first looked at the mean of T_R. The measurements clearly show that the T_R increases when the size of the object attribute increases and when the number of consumers increase, as shown in Fig. 4.11. The error bars indicate the standard deviations at each time step.

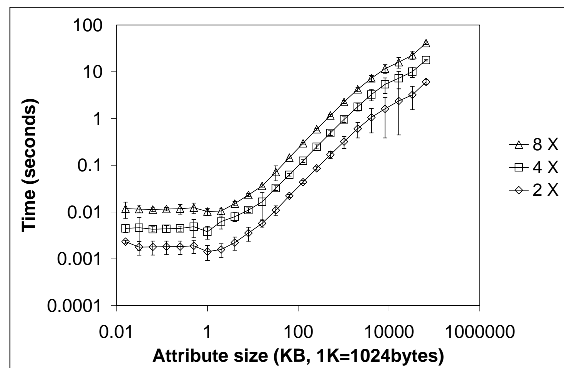


Figure 4.11: The comparison of the T_R.

By analysing the updating delay for each consumer, we found the remote update occurs in a one-by-one manner, which starts from the first consumer that joins the federation and ends with the last consumer, as shown in Fig. 4.12. An important reason for being so is that the RTI being used in the experiment does not support the multicast over TCP sockets. But the delay for the Nth consumer is smaller than N times the first consumer, as shown in Fig. 4.13. Because the local RTI library in the producer omits the operations for initialising remote updates when there are more consumers. The delay between the producer and the first consumer is consistent within the range of standard deviations, see Fig. 4.14.

4.4.5 Message passing

Messages are another type of data being exchanged between components. In ISS-Conductor, a typical message size is 90 bytes. The RTI of HLA treats the message distribution differently from the object distribution because of the non-persistency; but the underlying communication services are both based on sockets. Using the TCP sockets, messages are distributed to multiple receivers in a same manner as in data

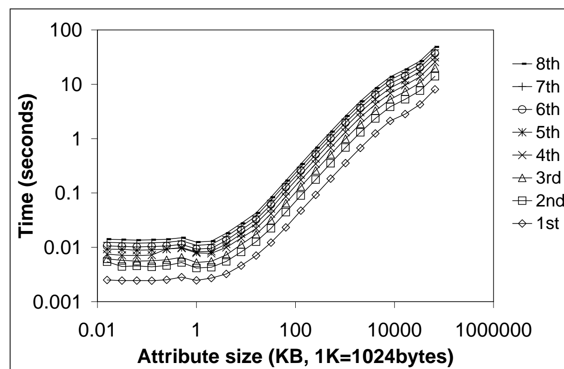


Figure 4.12: The remote update of all eight consumers.

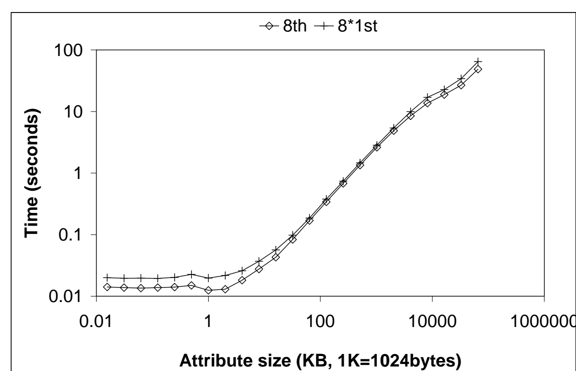


Figure 4.13: Compare the remote update delay of the 8th consumer and 8 times delay of the first consumer.

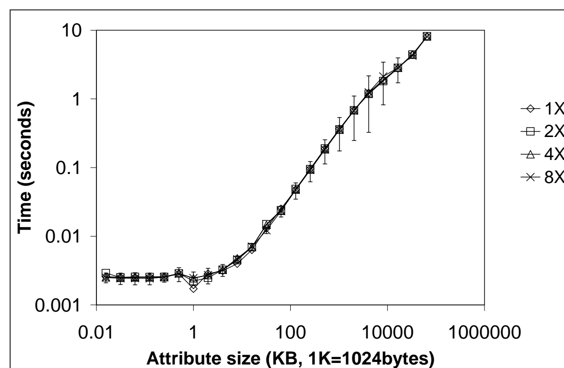


Figure 4.14: The remote update of the first Consumer in the federation in different configurations).

objects distribution. Fig. 4.15 shows the delay of sending a message to four receivers (each receive in a separate host).

We studied how a federate simultaneously handles both object distribution and message passing. A scenario, as shown in Fig. 4.16, is used in the experiment. Since we knew that both objects and messages are delivered sequentially, when the size of object is large, the message sent from Consumer A will arrive at the Producer A and

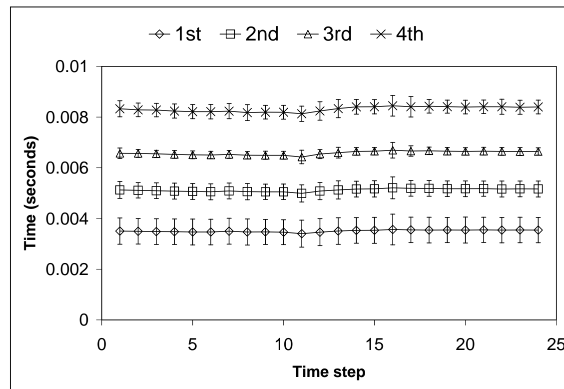


Figure 4.15: Passing messages to four receivers. The error bars indicate the standard deviations.

Consumer B when they are still sending and receiving respectively. We did the measurement in a same number of iterations as the previous scenarios. Fig. 4.17 shows the averages of 50 measurements. From the results, we see a federate has a delay in receiving income messages when it is sending large size data objects, but it has little influence when it is receiving data objects. The local RTI library of a federate handles the income messages before receiving the data objects, as shown by the curve in the Consumer B. This helps agents to respond to the incoming event in real time.

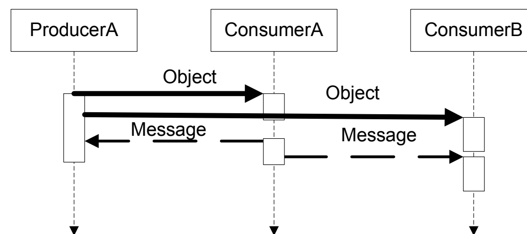


Figure 4.16: A scenario of sending data objects and messages between three component instances.

4.4.6 Object model and update delay

Finally, we studied the correlation between the structure of an object model and its remote update delay. In the experiment, we only focus on the object models with different number of attributes. We modified the object model in the previous experiment to have 16 attributes. And run the experiments in 4 configurations, in which the Producer updates 1, 2, 4, 8 and 16 attributes respectively and the attributes have equal sizes. In each configuration, the total size of the data attributes remains same. Fig. 4.18 shows the results of 50 measurements. We can see that the number of attributes does not influence the remote update of the object. It implies that the total size of the attributes is the main issue influencing the remote update delay.

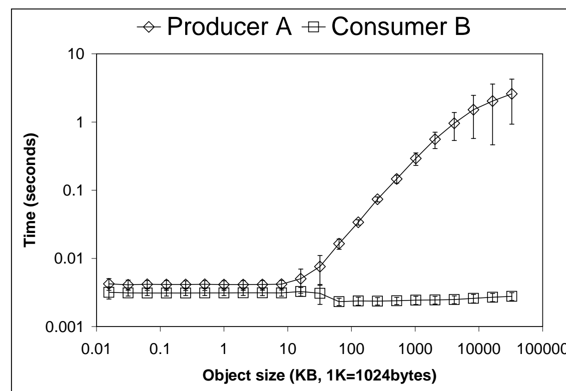


Figure 4.17: The influence between object distribution and message passing. The error bars show the standard deviation at each time step.

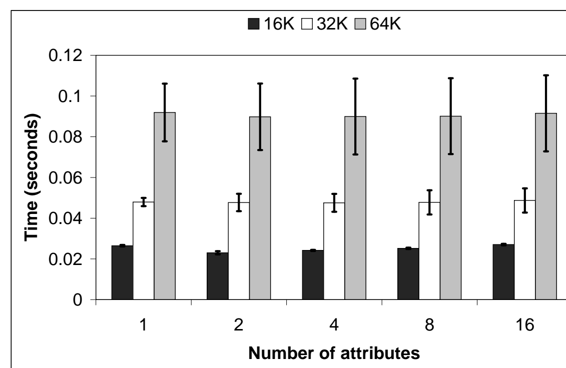


Figure 4.18: Remote update delay of different number of attributes. The error bars show the standard deviation at each time step.

4.4.7 Summary

The results of the experiments can be summarised as follows.

1. The remote update of a shared object depends on the size of the object and the number of the subscribers of the class, and does not have a clear dependencies on the location of the RTI execution and the number of attributes in the object.

2. For large size data objects, ComAs can achieve a comparable update delay to the normal TCP sockets. For small size objects, the latency of ComA is higher than the TCP sockets.

4.5 Performance for action reasoning and story execution

In this section, we focus on three main issues: first, the overall quality of the Module Agent, e.g. its correctness for making decisions on actions, second, the overhead of the

reasoning kernel, e.g. the delay for searching solutions, and third, a brief comparison of the performance of the three execution paradigms.
The benchmark story employs six roles, in which *Producer_A (PA), Producer_B (PB)*, and *Producer_C (PC)* are instances of the Producer, and *Consumer_A (CA), Consumer_B (CB)*, and *Consumer_C (CC)* are instances the Consumer. The story, as shown in Fig. 4.19, has three scenarios. The first scenario, called *Scenario A*, involves three roles: *Producer_A, Consumer_A*, and *Consumer_B*. The *Producer_A* exports data for the *Consumer_A* and *Consumer_B*, and continues after they finish the consumption, as shown in Fig. 4.20. The second scenario, *Scenario B*, involves *Producer_A, Producer_B*, and *Consumer_C*. In this scenario, *Producer_A* and *Producer_B* have critical transitions *SbT7* and *SbT8*, as shown in Fig. 4.21. And the third scenario, *Scenario C*, involves *Producer_A, Consumer_A, Producer_C*, and *Consumer_C*. This scenario has a dependency tree which is more than two layers, as shown in Fig. 4.22. The scenarios are switched in the sequence of *Start_Story, Scenario A, Scenario B, Scenario C* and *End_Story*. The *dataObj* is mapped to *Object1* in *Producer_A, Producer_B, Consumer_A* and *Consumer_B*, and is mapped to *Object2* in *Producer_C* and *Consumer_C*.

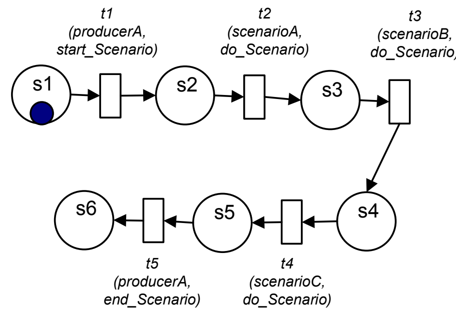


Figure 4.19: Benchmark story. It has three nested scenario nets: T2 (scenario A), T3 (scenario B) and T4 (scenario C).

The experiment has been executed in a single cluster of the DASII. Roles are executed in separate nodes; one role, both its Actor and Conductor, is executed in one node. The *producerA* is the responsible role for the initial scenario (scenarioA). At run time, ISS-Conductor generates a number of log files. The MAs generate a reasoning log file, which contains all the queries to the reasoning kernel, and the time cost for each query. We study the performance using these log files.

4.5.1 Overall observations on the action reasoning

From the perspective of the reasoning engine, all the facts of the *Capability* and *Story* and the rules for execution control are represented as Prolog terms. Table 4.2 shows a number of terms in the instances of Consumer and Producer.
The story is forced to execute in a distributed, hierarchical and centralised execution paradigm respectively. Each paradigm has been executed 20 times. From the experiment, we observed the following; first, the federates of all the roles (Actor and

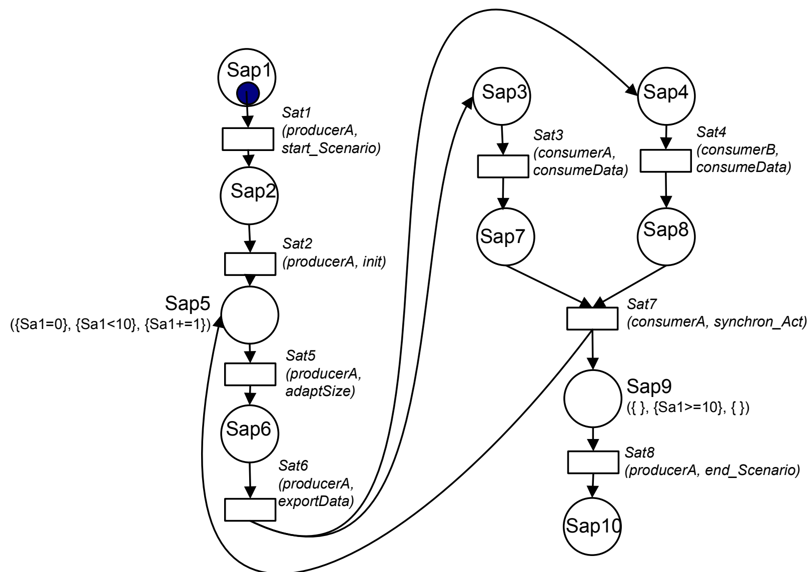


Figure 4.20: Scenario A (involved roles: Producer_A, Consumer_A and Consumer_B). Using the publish and subscribe mechanism, a data object can be simultaneously consumed by multiple consumers.

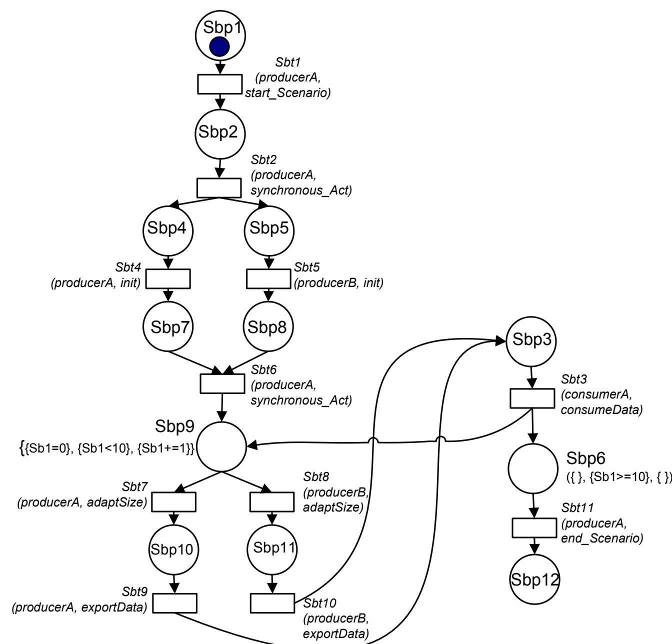


Figure 4.21: Scenario B (involved roles: Producer_A, Producer_B, Consumer_A). SbT7 *and* SbT8 *are two critical transitions.*

Conductor are two federates, and there are in total 12 federates) can successfully join and resign from the story federation. Second, all the Conductors can correctly find the actions for their Actor (by comparing the activity sequences with the scenario marking-graph). And finally, the roles in each scenario can successfully adapt their

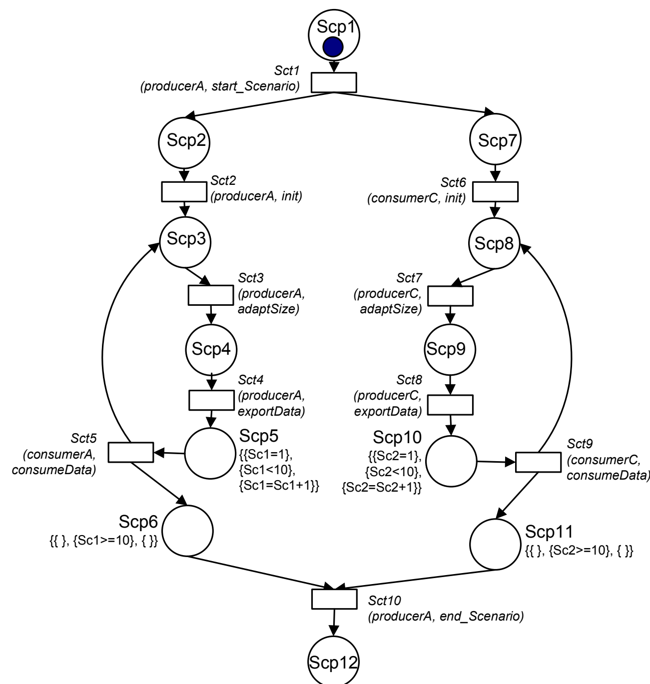


Figure 4.22: Scenario C (involved roles: Producer_A, Consumer_A, Producer_C and Consumer_C).

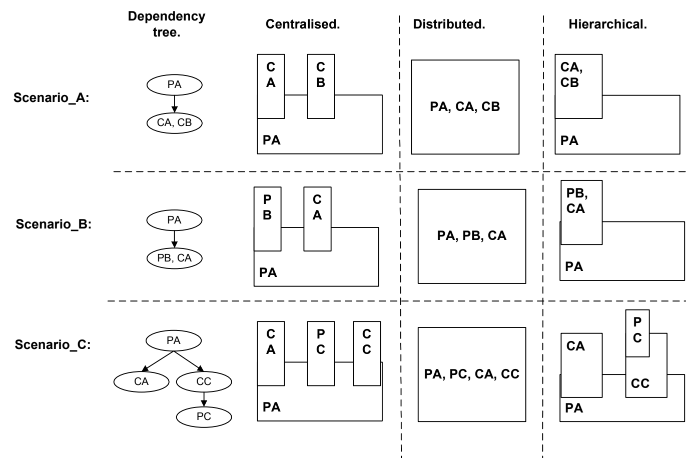


Figure 4.23: Topologies of the routing spaces of the involved roles. The roles that are not involved in the scenario set their routing spaces using the away from the others *profile.*

routing spaces, non-involved roles do not receive state information from the scenario, and involved roles can adapt their scenario in the correct way.

4.5.2 Overhead of the reasoning kernel

The overhead of the reasoning kernel is defined as the delays for processing the events that are received from the task interpreter, e.g. for reporting observations and for

Table 4.2: Facts and rules in the knowledge base of the Consumer and Producer components. The story facts and reasoning rules are consistent for all the component instances.

	Capability facts	***Story facts***	***Reasoning rules***
Consumer	20	154	380
Producer	28	154	380

querying new actions. The events for all three scenarios are traced in *Producer_A*; the total numbers and the time cost for each processing are shown in Fig. 4.24.

Execution paradigm		**Centralised**	**Distributed**	**Hierarchical**
Total events		6720(±2%)	5100(±2%)	4200(±2%)
Processing cost (seconds)	Min	<0.0001	<0.0001	<0.0001
	Average (Standard deviation)	0.002 (0.003)	0.002 (0.003)	0.002 (0.003)
	Max	<0.05	<0.05	<0.05

Figure 4.24: The total number of events and the time cost for processing an event.

In the centralised model, the co-ordinator has to search activities for all the other peers and maintain the states for them; therefore the total number of events is much higher than the distributed and hierarchical paradigms. In the hierarchical paradigm, irrelevant events are filtered, the total number of events is less than for the other two.
From the results, we can see the maximum delay for querying is less than 0.05 seconds, and the average cost is about 0.002 seconds which is comparable to the latency of the object distribution in ComAs (0.002 seconds). From this point, we can say that the reasoning kernel does not necessarily introduce a bottleneck of the system performance.

4.5.3 Reasoning complexity and delay

In this section, we briefly analyse the complexity of reasoning procedures in an MA and figure out the upper bound of the delays. In general, the reasoning engine accepts two types of requests from the task interpreter, see the architecture of MAs in Fig. 2.4: asserting observations to the world model, and querying activities for the Actor. The delay for the first type of operations is inherent to the implementation of the Prolog reasoning engine; the update of the dynamic database in the Prolog reasoning engine causes the overhead. The delay for the second type is related to the description of a *scenario net*, *capability*, and the number of involved roles. Since the operation for searching a doable action is frequently invoked in the second type for control run-time behaviour, we use the computing complexity of this operation to analyse the reasoning delay.

According to the algorithm we discussed in chapter three, searching a doable activity takes two steps. It first finds an *enabled transition* from a scenario net, and then finds a *doable* action from the capability for the transition. To find an enabled transition from a scenario net, the reasoning engine needs to traverse the transitions in a scenario net, and to scan the places in the pre set of a transition, therefore the complexity is $O(N_t \cdot N_p \cdot (N_{pg} + N_{tg}))$. N_p and N_t denote the number of places and transitions, N_{pg} and N_{tg} denote the maximum number of guide expressions in places and relation links, and N_r denote the number of roles. Finding a *doable* action in the capability can be two cases. If the action in the transition is *doable* return the action, the searching cost is merely for finding the fact from the dynamic database and evaluate the expression. Therefore the complexity is $O(N_a \cdot N_{ag})$, in which N_a and N_t respectively denote the number of activities and transitions in the capability, and N_{ag} denotes the maximum number of guide expressions in links. Otherwise the reasoning engine has to find another action in the capability which leads a path to the action; in Prolog, using a recursive algorithm to find the path between two nodes in a graph, the complexity is $O(P^T_{N_a} \cdot N_{ag})$, T is the possible intermediate actions in the path. In distributed and hierarchical paradigms, each role does all the search procedures; for the centralised paradigm, the co-ordinator not only searches actions for itself but also the *enabled transitions* for all the other involved roles, and the other peers only searches *doable* activities. We can consider the number of guide expressions as a constant, and the evaluation of the guide expressions is mostly inherent to the Prolog reasoning engine. The complexities of searching procedure are thus depicted in Table 4.3.

Table 4.3: Searching complexity of an activity.

Execution paradigm	**The best case**	**The worst case**
Centralised paradigm		
Co-ordinator	$O(N_r \cdot N_p \cdot N_t) + O(N_a)$	$O(N_r \cdot N_p \cdot N_t) + O(P^T_{N_a})$
Peers	$O(N_a)$	$O(P^T_{N_a})$
Distributed paradigm	$O(N_p \cdot N_t) + O(N_a)$	$O(N_p \cdot N_t) + O(P^T_{N_a})$
Hierarchical paradigm	$O(N_p \cdot N_t) + O(N_a)$	$O(N_p \cdot N_t) + O(P^T_{N_a})$

Using Prolog to implement the high level searching strategies, the programming itself is easier and flexible, but the price is the low efficiency of searching itself. In the analysis, we do not include the assumption on the order of clauses and other tricks, because most of them require knowledge on the capability and scenario net. From the analysis, we see that the performance can be principally improved in a number of ways. First, a sufficient detail scenario net makes the searching cost for *doable* action close to the best case; a small T means the reasoning engine does not need to find many intermediate activities. Second, a complex scenario net can be divided into a number of smaller sub nets, so that both N_t and N_p are small.

4.5.4 Brief comparison between execution paradigms

We study the execution paradigms by comparing their execution time for each scenario. Since *Producer_A* is the responsible role for all the scenarios, its time costs for each scenario when using different paradigm are compared. Fig. 4.25 shows the mean of the 20 executions.

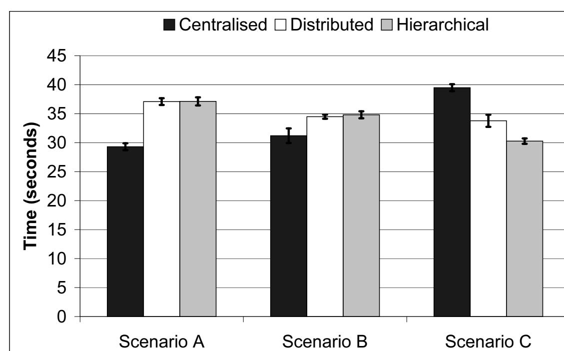


Figure 4.25: The time cost by Producer_A *for each scenario in different execution paradigms. The error bars show the standard deviations.*

From the results, we clearly see that the execution paradigms do influence the performance of the execution. When the scenario (scenario B) contains critical transitions, the centralised paradigm has a better performance than the other two. When the scenario (scenario C) has a dependency tree which has more than two layers, the hierarchical paradigm can get a better performance. An important reason for that is that when the dependency tree is deep the centralised paradigm does not exploit the parallelisation between the loosely dependent roles. It also introduces a load-balancing problem in the central co-ordinator; the central co-ordinator has to do many more queries than the other two paradigms. That is also the reason that when the number of roles is small as in scenario A, the centralised paradigm can achieve a better performance than the other two.
Between the distributed and hierarchical paradigm, the performance difference is not remarkable when the dependency tree is only two layers, as in scenario A and B. We can clearly see that the hierarchical paradigm reduced the number of messages in scenario C, as shown in Fig. 4.26.

4.5.5 Summary

From the experiments, we can say the current implementation of the reasoning kernel can correctly realise the interpretation of a story, and the MAs can successfully control the system execution. The system performance is influenced by the design of the scenario and the selection of the execution paradigm. The reasoning kernel has a comparable latency to the ComAs, and is not necessarily a killer for the system performance.

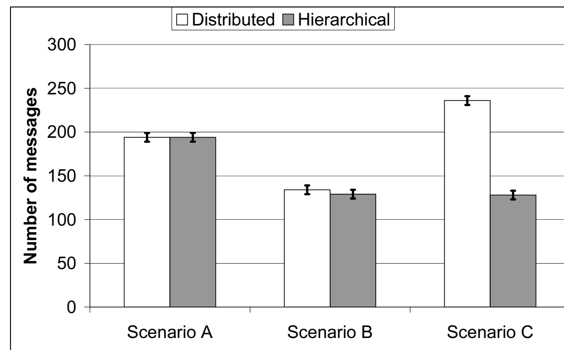


Figure 4.26: The total number of state-update messages received by the Producer_A *in each scenario.*

4.6 Discussion and conclusions

4.6.1 Evaluation

In this chapter, we have discussed the implementation of Communication Agents and Module Agents and have studied the associated performance issues. In ComAs, the services provided by the RTI of HLA are used for the underlying data communication. In the MAs, the control intelligence for activity control and decision making is realised using a logic language (Prolog). Its reasoning engine supports the solution searching and maintenance of dynamic databases.
An original goal of ISS-Conductor is to provide an architecture which can *efficiently interconnect simulation and visualisation programs, and can support the rapid prototyping of interactive simulation systems*. We have not deployed ISS-Conductor in real cases of simulation and visualisation systems. Yet, from the experimental results presented in 4.4 and 4.5, we can evaluate the development from following aspects:

1. We have implemented the agent framework discussed in chapter two. By constructing the interaction story of the test case, we see that using the ISS-Conductor components, the system behaviour can be adapted at the story level and does not demand modifications to the component kernels. The logic of the system behaviour can be adapted at the story level.

2. We have realised the story execution mechanisms discussed in chapter three. In 4.5, we have executed the scenarios in distributed, hierarchical and centralised modes. From the experiments, we see that the current implementation of the reasoning kernel can correctly realise the interpretation of a story, and the MAs can successfully control the system execution.

3. In the experiments, we have also studied a number of performance characteristics of the implementation. The Communication Agents add limited overhead on the data transmission; in general, the latency of transferring messages or small size objects is acceptable for *soft* real-time interaction§. The ComAs achieve a

§As we have mentioned, the DoD's RTI does not support *hard* real-time distributed simulations [67].

comparable throughput to pure TCP sockets when transmitting large size objects. Of course, the transmission can be optimised at application level; in the next chapter we will show how parallel data producers improve the transmission delay.

From the experimental results, we can say that the implementation of ISS-Conductor fulfils the basic design requirements.

4.6.2 Conclusions

The discussion leads following conclusions:

1. High Level Architecture (HLA) provides a flexible interface for implementing ComAs. The RTI services provide a standard way for accessing and updating shared objects, and for adapting the multicast group between the roles. The RTI services can support the real-time interaction between ISS modules.

2. Separating the control of the run-time interactions from the functionality of the system modules improves the reusability of the constituent system components and the adaptability of the overall system behaviour.

3. By benchmarking the Module Agents, we see that the reasoning delay of Prolog is comparable to the communication latency in the test case. By analysing the reasoning complexity of the implementation, we can see the performance is dependent on the complexity of scenario net, and the scenario net can be simplified by dividing into smaller sub nets. Therefore, the reasoning is not necessarily the bottleneck for the system performance.

In the next chapter, we will use a test case to demonstrate the main features of ISS-Conductor.

Chapter 5

Rapid Prototyping of a surgical pre-operative planning environment

5.1 Introduction

Making an optimal plan for a vascular operation is difficult, not only because locating and analysing the affected vessels is time consuming but also because the surgeon must consider the effects of the operation on the other possible diseases of the patient. Using computers to simulate the surgical procedures and to evaluate their effects is considered to be an important aid for pre-operative planning [144–146]. However, the complexity of developing such simulation systems and the very high requirements on system performance and real-time interaction hamper their introduction. In this chapter, we use the ISS-Conductor architecture for rapidly prototyping an adaptable environment for planning vascular operations*.

5.1.1 Background

Vascular disorders, such as stenosis or aneurysms†, can cause serious diseases due to their influences on the blood flow; improving the flow quality in the affected vessels is the basic approach to treat these disorders. Vascular reconstruction is a surgical procedure which redirects the blood flow from the affected area using a grafted bridge, also called a *bypass*. It is applied when less invasive treatments, e.g. thrombolysis and balloon angioplasty [149] are not an option.

*Parts of this chapter have been published in *Z. Zhao, R. G. Belleman, G. D. van Albada and P. M. A. Sloot. "AG-IVE* an agent based solution to constructing Interactive Simulation Systems" in the proceedings of the second International Conference of Computational Science (ICCS02), Amsterdam, NL, 2002.

†A stenosis is a obstruction or narrowing of the artery by the accumulation of fat, cholesterol and other substances in the vascular wall. Aneurysms are a ballooning out of the wall of an artery due to a weakness in the wall [147, 148].

In vascular operations, to optimally place a bypass, one has to consider not only the structure of the affected artery but also the actual improvement in the blood flow. To plan an operation, a surgeon first needs information about the location and the structure of the affected artery. Medical imaging techniques, e.g. X-ray angiography, computed tomography (CT) or MRI (magnetic resonance imaging), can be used to obtain digital images of the vessel structures, which can be represented intuitively. After that, a plan is made based on the analysis provided by radiologists. A number of simulation or visualisation based tools can be used to aid the design and evaluation of a plan. The first of these would be a tool for analysing medical images, with which a user (surgeon or radiologist) can locate and segment the information of affected vessels from the raw scanned images. An interactive visualisation for representing segmented information as 3D objects and for prototyping trial bypasses would be next. A simulator for computing properties of the blood flow in vessels is desirable for evaluating the actual effect of a bypass. A possible scenario for deploying these tools in operation preparation is shown in Fig. 5.1.

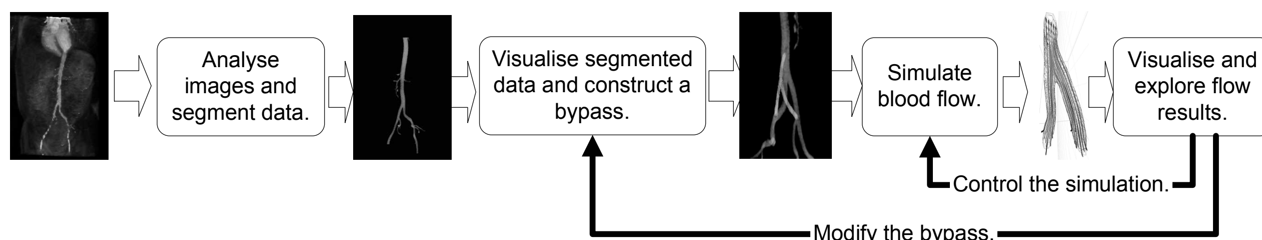


Figure 5.1: A scenario of simulation based operation planning.

During the past decade, the development of these tools has attracted a great deal of attention from both the Medical and Computational Science communities [150–152]. A series of tools for medical image processing and visualisation [153,154], and for simulating blood flows [155] have been developed. Using these existing tools, a synthetic environment for making a surgical plan and for simulating activities in an operation theatre may be developed. Compared to using standalone tools, an integrated environment has a number of advantages. First, it allows a surgeon to tune the structure of a bypass in the run-time loop of the blood simulation, which can not only improve the efficiency for bypass refinement, but also save the resource consumption for both computation and storage. Secondly, by coupling these tools together it becomes possible to mimic the actual activities in an operation theatre, which is very useful for surgeons.

The Section Computational Science (SCS) at University of Amsterdam (UvA) [156] has been specially interested in simulating blood flows and in building virtual reality based environments for exploring medical images. A number of simulation and visualisation packages were produced. In this chapter, we use some of these packages as basic material to prototype an interactive environment for simulation aided operation planning, and thus show the use and limitation of the ISS-Conductor architecture as a rapid prototyping environment.

5.1.2 Goal of the chapter

Two packages: a flow simulator (*Flow_Simulator*) and an interactive visualisation tool (*Desktop_VRE*) are selected as basic material for demonstrating the deployment of the ISS-Conductor architecture:

1. *Wrapping legacy systems as software components*. The first feature provided by ISS-Conductor is to encapsulate legacy systems as reusable components. In section 5.2, we explain the detailed procedures for incorporating a fluid-flow simulator and an interactive visualisation program into the ISS-Conductor architecture.

2. *Coupling components to create an Interactive Simulation System*. An ISS is constructed using components. At run time, the Communication Agents realise the basic interconnection between component instances, and the Module Agents control their scenario specific activities. In section 5.3, we discuss the basic steps for coupling the components.

3. *Adaptable interaction*. The behaviour of an ISS can be adapted by modifying the activity constraints in the story. In section 5.3.4, two scenarios are used to demonstrate this feature.

4. *Including application specific control intelligence*. Without changing the implementation of the components, an ISS developer can include application specific control intelligence in the knowledge base of agents. In section 5.4, we use an example to demonstrate it.

5. *Supporting problem solving*. Using ISS-Conductor, an ISS can be promoted to support problem solving, e.g. collaborative solution searching, at the system behaviour description level. In 5.5, we use an example to describe its realisation.

We demonstrate these features and discuss the experimental results for a number of focal points related to the implementation quality of ISS-Conductor. The first focus is the development costs, when using ISS-Conductor for wrapping simulation and visualisation programs and for coupling them into an ISS, the second focus is the remote update delay for simulation results, and the third one is the scalability of an ISS when it supports collaborative interactions. There are a number of reasons for choosing these issues. First, rapidly prototyping human-in-the-loop simulation based experiments is one of the original goals of developing ISS-Conductor; reducing the development costs for an ISS is a necessary promise of the implementation. Second, updating the simulation results between distributed modules is critical to the system performance, both for refreshing visualisation scenes and for human interaction. Finally, the RTI of HLA claims to be a scalable software; ISS-Conductor complements the basic HLA services with high-level support for controlling interaction scenarios, thus, its influence on the system scalability is also an important issue.

5.2 From Legacy systems to reusable components

In this section, we shall discuss the basic procedures to incorporate a legacy simulation or visualisation program into the ISS-Conductor compliant architecture. A simulation and a visualisation are used as examples.

5.2.1 Basic steps

Incorporating a legacy simulation or visualisation system into the ISS-Conductor architecture takes three main steps: 1) defining the capability of the component, 2) adapting the source code into the required style and 3) generating the executable component.

Defining the component capability

The capability of a component is defined based on the analyses of the documentation of the legacy system and of the requirements of the component.

1. *Defining data classes*. The data structures defined in the legacy systems are described as *data classes*; the classes to be used only by the component are defined as *internal classes*, others are defined as *shared classes*. The definition of the *data classes* can be documented using OMT [48] based templates (see section 4.3.3).

2. *Defining actions*. The actions of the component are defined based on the actual functionality of the legacy system and the desired services that the product component intends to offer. The data dependencies of an action are described as two lists of *data classes* for indicating the input and output requirements respectively. The actions contain an initial and one or more terminal actions. The final execution states of an action are defined based on the possible execution output.

3. *Describing action dependencies*. The dependencies between actions are described based on the control and data flow in the legacy system.

Finally, using the capability template (see section 4.3.3), a capability specification is produced.

Incorporating source code

A source framework can be automatically generated according to the capability specification. The framework contains an interface for wrapping the legacy assets as an Actor, and code for generating a Conductor. The component developer needs to associate the legacy routines and data structures with the wrapper interface of the Actor.

1. The variables defined in the legacy system are directly associated with the attributes of the internal data objects in the Actor. The association has to take into account the consistency between the lifecycle of the data objects and the scopes of the variables.

2. In the Actor, the initialisation of the ComA is related to the original execution style of the legacy system: sequential, multithread or multiple processes. A process can only contain one ComA. If it is a multithread system, the ComA is incorporated as a separate thread. If it has multiple processes at run time, each process has its own ComA.

Generating the executable

The final step is to generate the executable of the component; it includes a capability specification and binaries of Actor and Conductor. Without supplying any *stories*, a component can be executed in a *debug* mode. In the debug mode, the Conductor generates a dummy story to invoke the actions in the Actor and test their possible transitions. The component generates a number of log files, which can be used by the developer to debug the implementation.

5.2.2 Legacy flow simulation and visualisation systems

The *Flow_Simulator* can simulate blood flow using a given geometrical boundary. The setting of the simulation is passed to the program through a configuration file. The computing kernel uses the lattice-Boltzmann method [157] and is written in C. The program is parallelised using the Message Passing Interface (MPI) [119]. Fig. 5.2 shows its basic functionality. The program first checks the validity of the input setting before the simulation, it has a routine for exporting intermediate computing results to data files. The end condition of the computation is controlled using a maximum iteration number.

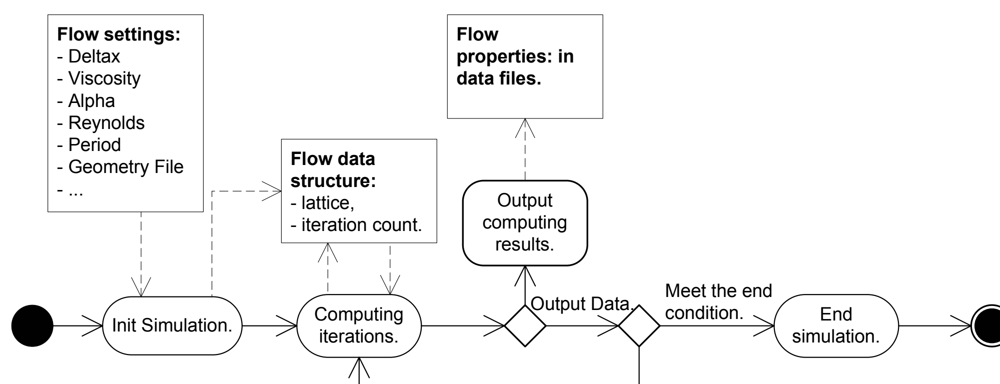


Figure 5.2: The basic functionality of Flow_Simulator.

The *Desktop_VRE* system is derived from an earlier system, named the Virtual Radiology Explorer (VRE) [158], which was originally developed for an immersive virtual-

reality environment, the CAVE [159]. The *Desktop_VRE* system ports the basic functionality of the VRE system and realises it on normal desktops. It allows a user to compose geometrical structures for doing flow simulations. The visualisation kernel is implemented using the Visualisation Toolkit (VTK) [160], and the basic data structures are in VTK formats. Fig. 5.3 shows the main features of the *Desktop_VRE* system.

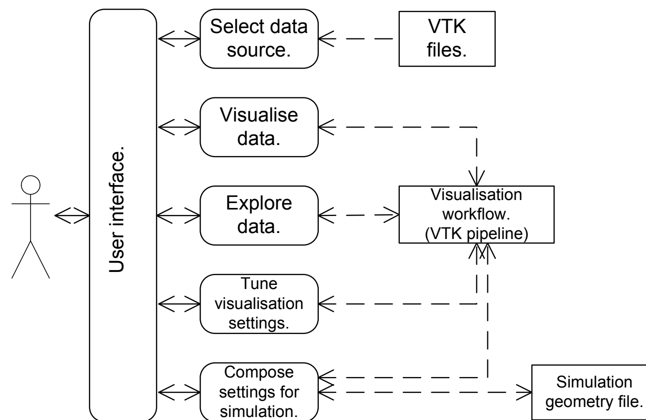


Figure 5.3: The basic functionality of Desktop_VRE.

Following the basic steps, the two legacy systems have been incorporated as two components: *C_Flow_Simulator* and *C_Desktop_VRE*.

5.2.3 Component 1: *C_Flow_Simulator*

Capability description and incorporation

The data structures and variables in the *Flow_Simulator* are grouped into three data classes. An internal class called *Status_Monitor* encapsulates the variables for controlling computing loop and simulation states. Two shared classes encapsulate the setting for the simulation and the properties of fluid flow, called *Flow_Setting* and *Flow_Output* respectively. The basic functionality of the legacy system is described using eight actions: *Start*, *Get_Simulation_Setting, Set_Default_Setting, Init_Simulation, Export_Flow, Compute, Rest*, and *Quit*, in which *Start* and *Quit* are the initial and terminal activities respectively. According to the execution branches of the *Flow_Simulator* implementation, two finished states are defined for describing the execution: *succeed* and *failed*. Fig. 5.4 shows the dependencies of these actions.

Overhead on computing

The *Flow_Simulator* is a parallel program, so each process is equipped with a ComA. The data and actions are wrapped in the Actor. Using the debug mode, we measured the time cost for computing an iteration in both *Flow_Simulator* and *C_Flow_Simulator*. The *compute* action in *C_Flow_Simulator* is imported from the computing loop in

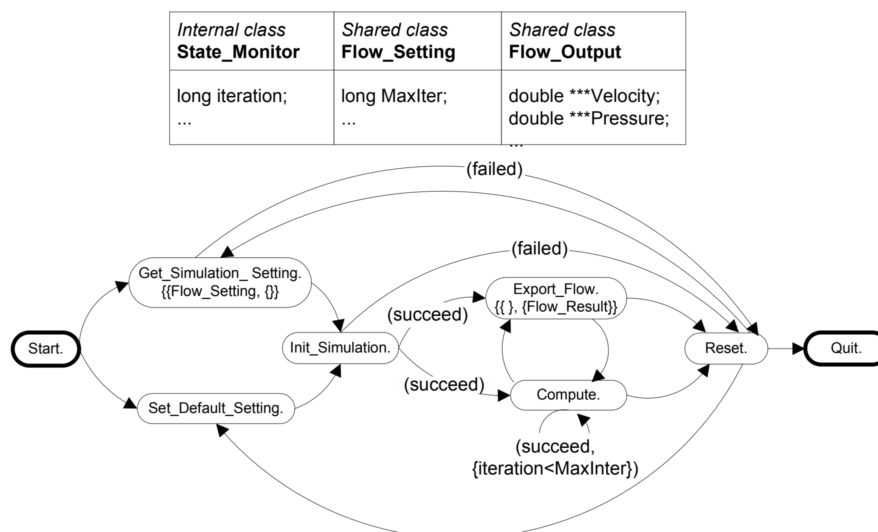


Figure 5.4: A partial activity-transition graph of the C_Flow_Simulator component. See the definition of component capability in section 3.2.

Flow_Simulator; some overhead may be introduced by the processing of ComA events. Fig. 5.5 shows the time cost for an iteration when using a tube with $32 \times 32 \times 64$ lattice units as the geometrical structure. The results are the average of 1000 iterations. From the measurements, we can observe the overhead of the ISS-Conductor, especially when the number of processes is small. We can see that the computing cost for each iteration decreases when the number of processes increases; but due to the increasing costs for inter-process communication, the standard deviation for the computation also increases. From the comparison, we can see that the performance of the ISS-Conductor component is reasonably close to that of the legacy implementation.

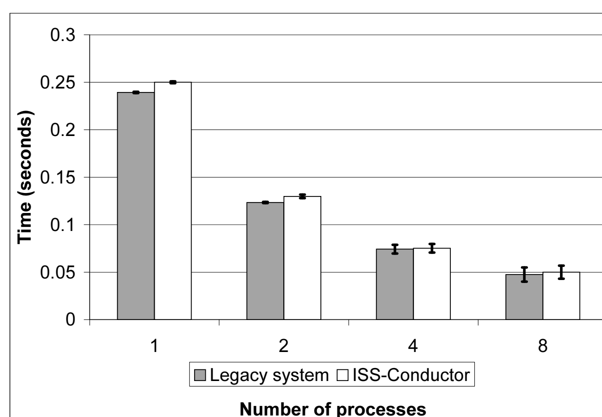


Figure 5.5: Performance comparison between ISS-Conductor component and legacy implementation. The measurement shows the time cost for one iteration. The error bars indicate the standard deviations.

5.2.4 Component 2: *C_Desktop_VRE*

Capability description and incorporation

Similar to the *C_Flow_Simulator* component, the *C_Desktop_VRE* component also has three data classes: an internal class, which encapsulates the variables for controlling the visualisation pipeline and the user activity states, and two shared classes which encapsulate the input flow data and the output of the flow boundary, called *Flow_Data* and *Flow_Boundary* respectively. The *Desktop_VRE* program is a human-centred interaction system. In Chapter three, we discussed an activity-theory-based layered model to describe the capability of such systems (see section 3.2.2). The functionality of the *Desktop_VRE* system is modelled as four main tasks: composing a simulation setting, visualising flow data, selecting tasks and exiting the execution. The subtasks for each task are used to model the user activity states. The operations for each subtask are mapped to corresponding elements in the user interface. Fig. 5.6 shows the layered picture of the functionality. Fig. 5.7 shows a partial activity transition graph of the capability.
In the activity transition graph, the actions to handle the input and output of shared data objects are also included, e.g. *Refesh_Flow_Data* action. The data classes appear in the pre or post condition list of the action, and the transformation between ISS-Conductor data objects to application specific data format are also implemented.

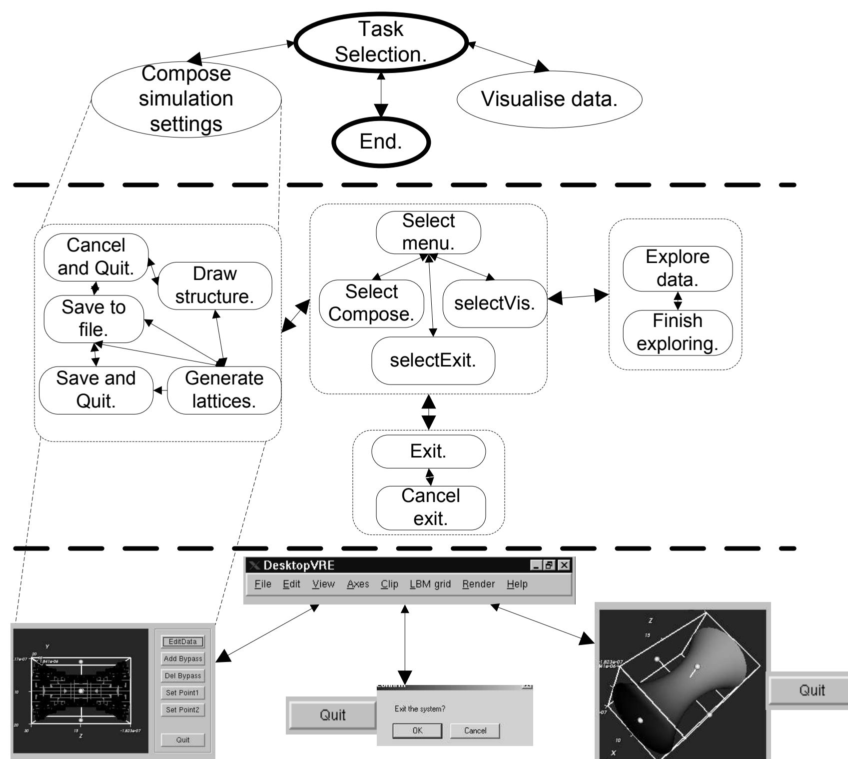


Figure 5.6: A layered vision of the functionality of the Desktop_VRE system.

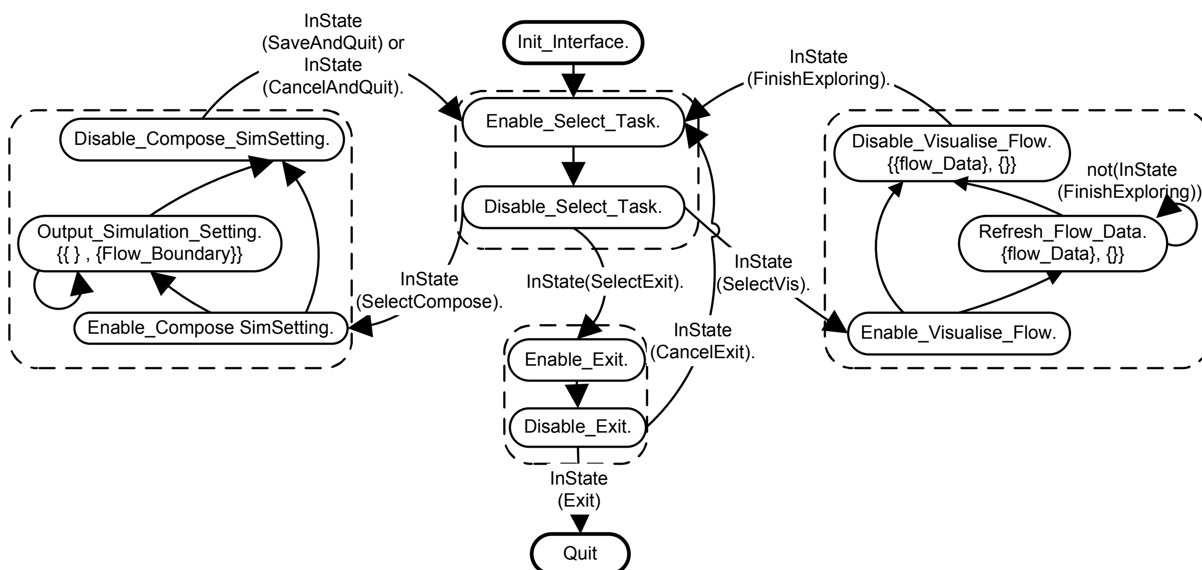


Figure 5.7: A partial activity-transition of the C_Desktop_VRE component. The term InState *describes the user activity state.*

Latency of state update

The Conductor of a component can be executed in an interactive mode. From the user interface of the Conductor, the user can see the world model and other run-time information of the Actor. The delay for perceiving user activity is critical for the Conductor to make decisions on controlling component behaviour. We measured the latency for the Conductor to perceive the states of user activities. The latency is defined as the delay from the user interactions with an interface element, e.g. clicking a button, until the Conductor perceives it. The experiment is performed on two separate nodes in DAS II supercomputer. The average latency is about 0.002 seconds, which is close to the latency for passing an HLA message.

5.2.5 Discussion

In this section, we have discussed the basic procedures to incorporate a legacy system into the ISS-Conductor architecture, and demonstrated them using two existing systems. These two legacy systems are well documented, the incorporation of the two components took in total 40 working hours. Half of the time was spent in defining the data classes and activity transition graphs. Since the state and action names appear as normal strings in the code, any misspelling cannot be checked at compiling time, which is inconvenient for debugging.

5.3 Coupling component instances

The components are deployed for rapidly prototyping an interactive simulation system. The main goal is to demonstrate the development of an interactive story, and the adaptability of the system behaviour. We start from a simple scenario, called *Blood_Flow_Studying*, in which a surgeon studies the properties of blood flow in a given bypass using a live flow simulation.

5.3.1 Basic analysis: roles and interactions

In the *Blood_Flow_Studying* scenario, components take two roles: one for simulating blood flow noted as *Blood_Simulator* and one for simultaneous presentation of the flow data called *Surgeon*, which are instances of the *C_Flow_Simulator* and *C_Desktop_VRE* components respectively. For the moment, we assume these two components are qualified for these two roles. In the next chapter, we will have more discussion on component selection and composition. The interaction between the roles can be described using an activity diagram, as in Fig. 5.8. In the diagram, activity states are the actions defined in the capabilities of the components. The data classes are mapped to shared data classes defined in each component respectively. The conditions at the decision point are described using the states of user actions.

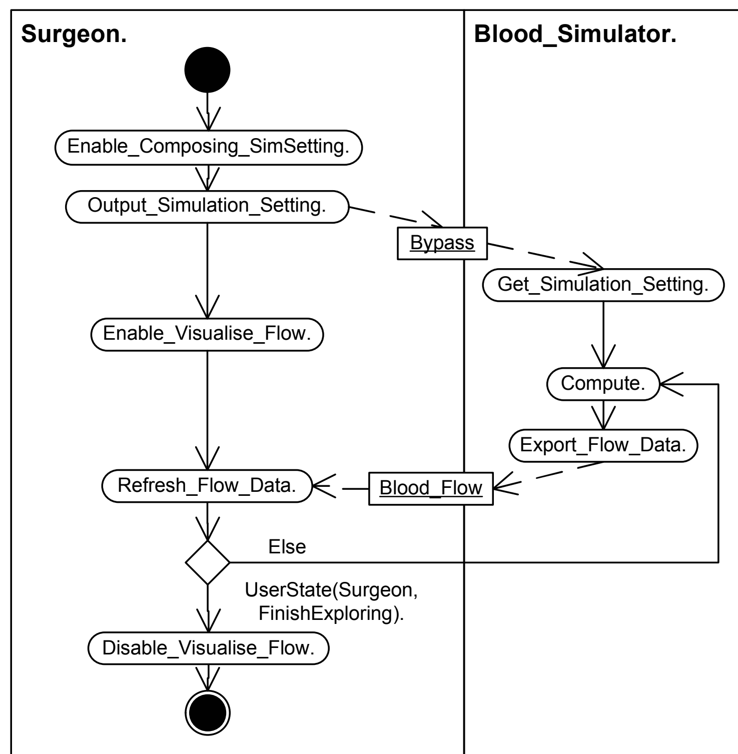


Figure 5.8: An activity diagram for Blood_Flow_Studying scenario. The term UserState *describes the activity state of a user.*

5.3.2 Making an interaction story

An interaction story contains three main parts: a common data interface between roles, one or more scenario nets for describing their activity dependencies, and run-time requirements for generating execution scripts. A common data interface defines an object model for different roles to exchange their data. In HLA, such object models are also called the Federation Object Models. Since mapping between data classes is only syntactic, the semantic level checking has to be done in the design stage. A scenario net can be derived from activity diagrams. In the *Blood_Flow_Studying* scenario, the branches in the decision point are based on the user's activity. If the user is in the *FinishExploring* state, the scenario finishes. The user's activity state is monitored by the ComA coupled with the GUI of the legacy system.

ISS-Conductor provides interface for describing guard conditions between activities. Fig. 5.9 shows a scenario net of the scenario. The story file contains descriptions of the required resources, e.g. number of processors and computing hours. This part of the description are used to generate scripts for different job submission tools, such as Portable Batch System (PBS) [161].

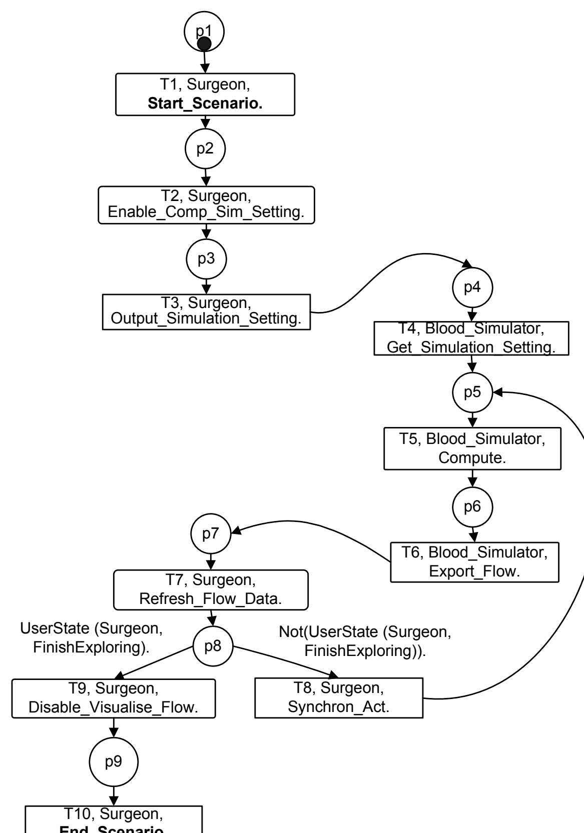


Figure 5.9: A scenario net of the Blood_Flow_Studying *scenario.*

5.3.3 Executing an ISS

At run time, a component is initialised as a role by assigning a name and a story. The Actor and Conductor of a role can be executed on different machines. The responsible role of the scenario is loaded later than the other roles. Since the update of simulation states (computing results of the flow) is critical for the human interaction, we studied a number of related performance characteristics using the example.

Execution

The system is executed on a single cluster of the DAS II supercomputer. The Actor of the *Blood_Simulator* is submitted using the open PBS tool, and its Conductor is executed on a separate node. The Conductor and Actor of the *Surgeon* are executed on separate nodes. The RTI is executed on a file server of the cluster. From the experiment, we see, all the ComAs can successfully join in and resign from the federation. The *Blood_Flow* object can be correctly visualised in the interface of *Surgeon*. Fig. 5.10 shows a screen snapshot.

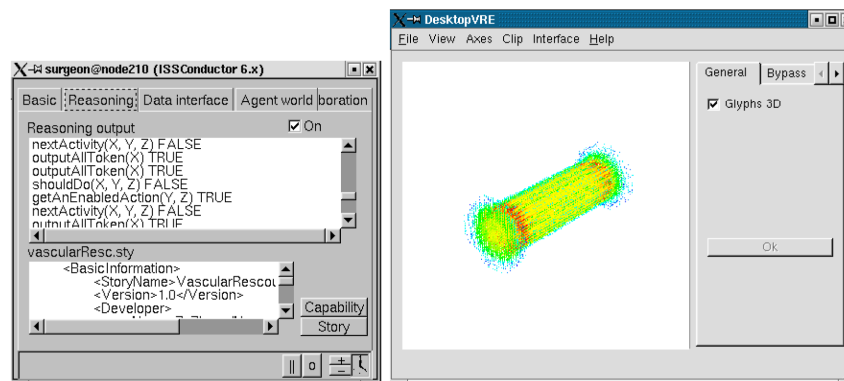


Figure 5.10: A screen snapshot of the Blood_Flow_Studying *scenario. The left window shows the interface of the Conductor and the right one is the interface of the* C_Desktop_VRE *component. The flow boundary is a tube with 32×32×64 lattices. The image in the window shows the velocity vectors of the calculated flow field.*

Remotely updating multiple shared objects

When executing the *Blood_Simulator* role in parallel, each process maintains only a part of the simulation results as different instances of the *Blood_Flow* class. These instances have to be merged before being visualised by the *Surgeon*. The merging can take place on either side. When the simulation is more compute intensive than the visualisation module, the second strategy is preferable, it is adopted in the current implementation.
In the scenario, the geometrical structure of the *Bypass* object is a tube. In the experiment, three different sizes of tubes are used, which are 32×32×16, 32×32×32 and

$32 \times 32 \times 64$ (lattice units)[‡].

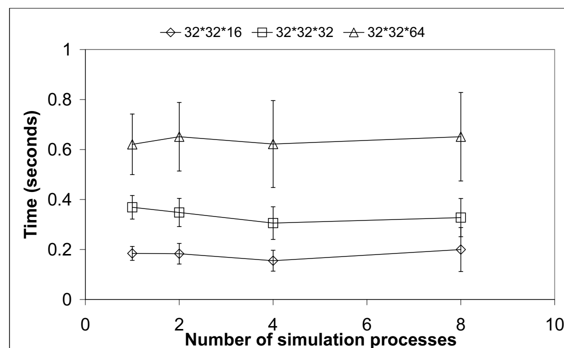


Figure 5.11: Remote update of shared objects when the simulation is executed on multiple processes. The error bars indicate the standard deviations of 100 measurements.

The delay of updating a *Blood_Flow* object is measured as the interval between the moment that the first simulation process starts to update its object and the moment that the visualisation component finishes its reflection of the object from the last simulation process. Each experiment is performed using four configurations: the simulation is executed in one, two, four and eight processes respectively. Fig. 5.11 shows the mean of the 100 measurements, and the error bars indicate the standard deviations. From the results, we can see the delay for remotely updating and reflecting multiple objects increases with the size of data objects, but within the standard deviation, it is independent of the numbers of processes. When the number of simulation processes increases, the size of the data object maintained in each process decreases but the total volume of the data objects remains same. At run time, the simulation processes can update the data objects simultaneously, but the RTI call-back function for reflecting the updates can only handle them sequentially, therefore the total update delay remains constant.

5.3.4 Asynchronous data update

From the execution of the *Blood_Flow_Studying* scenario, we see that the *Blood_Simulator* role only continues its *Compute* action after the execution of action *Refresh_Flow_Data* in the *Surgeon* role, which means the simulation is paused while the data is being visualised. When the simulation has a large number of lattice points, this is inefficient. One way to improve it is to allow the *Blood_Simulator* role to compute asynchronously with the *Surgeon* role. We call the new scenario as *Blood_Flow_Studying_Asy*. In the *Blood_Flow_Studying_Asy* scenario, the decision point for continuing computing does not directly depend on the execution of *Refresh_Flow_Data* activity. Fig. 5.12 shows the scenario net.

As we mentioned in the chapter 3, the updates of a shared object are buffered by the ComA before being processed. The buffer size can be customised at run-time. In

[‡]The size of the objects when using these three geometrical structures are 512KB, 1MB and 2MB respectively.

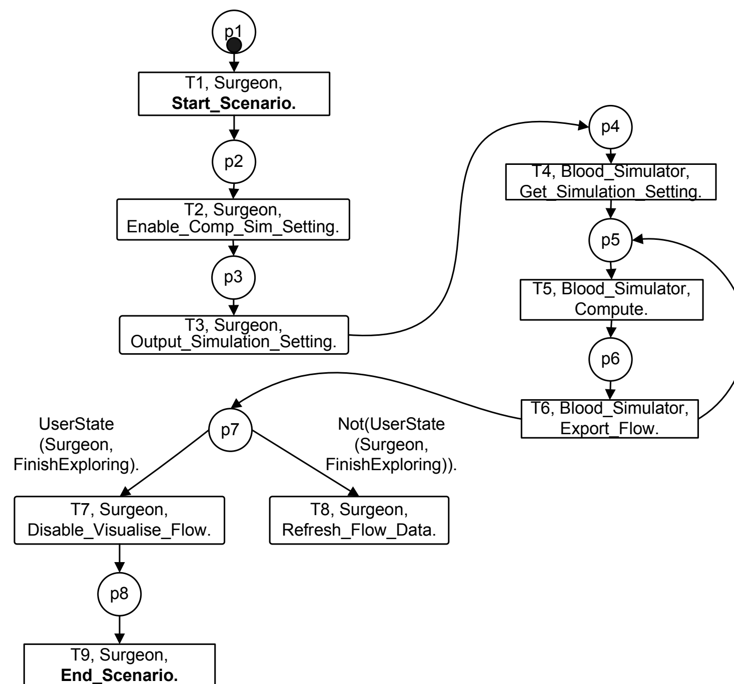


Figure 5.12: Asynchronous data transmission between the Surgeon_A *and* Blood_Simulator *component instances.*

the asynchronous scenario, we set the buffer size as the maximum, therefore all the simulation results can be visualised if even there is no synchronisation check.
We compared these two scenarios. At run time, each invocation of the *Compute* action calculates 20 iterations. The time intervals between two invocations of the *Compute* actions are measured. Fig. 5.13 shows the mean of 50 measurements when *Flow_Simulator* role in each scenario is executed in 1 and 2 processes. We can clearly see an improvement in asynchronous case.

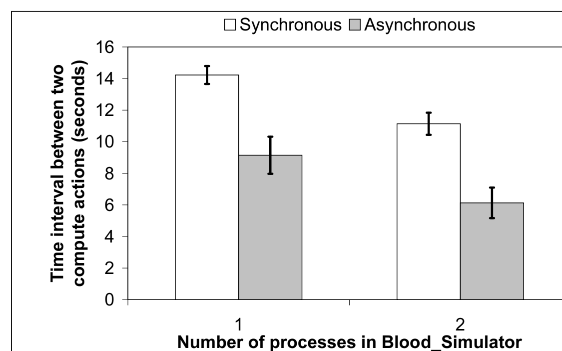


Figure 5.13: A comparison of the time interval between two invocations of Compute *actions.*

5.4 Automatic tuning of service quality

Optimising the performance of constituent components and improving the overall service quality of the system is another important concern in the system development. In this section, we discuss how the performance optimisation is included in the prototype ISS.

5.4.1 Adaptable state update

The visualisation of the simulation results is an important quality attribute of an ISS, which refers not only to the quality of the data presentation, but also to the update of dynamically changed simulation results in the visualisation pipeline. The user prefers to see a smooth change of the simulation states. In the interaction scenario discussed in the previous section, the *Blood_Flow* is preferably available whenever the *Surgeon* role needs to invoke the *Refresh_Flow_Data* action, and it has a minimal number of un-processed data in its input buffer[§]. From the discussion in the previous section, we see that neither of the update paradigms can surely meet this requirement. In the synchronous scenario, computing and visualisation actually work in a sequential way, and the delay for updating visualisation scene is not only introduced by the visualisation itself but also by the computing in the simulation. In the asynchronous scenario, the rate of exporting simulation results is only dependent on the speed for doing the *compute* action, and cannot guarantee the *Refresh_Flow_Data* action always gets object instance when it needs.
Fig. 5.14 shows the basic activities and the time costs for updating a *Blood_Flow* object. The total delay is the sum of a number of individual delays: computing (T_c), exporting data (T_e), receiving data (T_r) and refreshing the visualisation and rendering (T_R). T_f and T_m are relatively small and remain independent with the size of the object[¶]. T_c is related to the number of the lattice points in each simulation process and the number of the iterations in each cycle; T_e, T_t, and T_r are dependent on the volume of the data; and T_R is not only related to the data volume but also the algorithms for visualisation and rendering. Belleman gave a detailed discussion on these issues in his Ph.D thesis [12].
The performance of the object update can be improved using a number of techniques, such as applying dedicated algorithms to accelerate the computing action [155, 162], choosing efficient representation for data visualisation [34, 163], or reducing the volume of the data object [164]. However, the requirements discussed above still cannot be met by only applying those techniques. One of the reasons is that these techniques mainly aim at improving the performance of a single component, which do not consider the run-time interaction with the other related components. Traditionally, the control parameters of different components are statically configured in the coupling solution, which does not include the consideration of the run-time service quality of

[§]The updates of a shared object is buffered in the ComA of the receiver role.
[¶]We can see this from the experiments in the previous chapter.

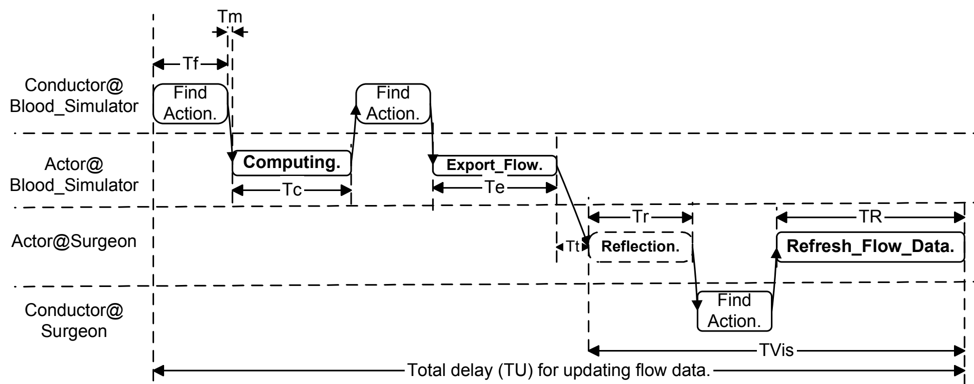


Figure 5.14: Basic time costs for updating a Flow_Data *object.*

the computing environment. In the ISS-Conductor architecture, a framework is provided for adapting the service quality of components and for optimising the overall system performance.

5.4.2 Solutions in ISS-Conductor

In ISS-Conductor, the quality attributes of the services and the control parameters that can be used to tune the service quality are explicitly described in the component capability. In the story, the constraints between the service attributes of each component are described as the requirements on the global system service quality. The services provided by an ISS-Conductor component include its activities and the data objects it produces. At run time, the ComAs in a component monitor the service quality, and the MA in the Conductor propagates the observations of actual service quality to the other MAs and can tune the control parameters according to the performance requirements. In a scenario, only one MA at one time is allowed to evaluate the constraints and to modify its parameters; a control token is used to co-ordinate the procedure.

5.4.3 An example: adaptable rate for exporting *Flow_Data*

In the system, the computational costs for T_c, T_e, T_r and T_R are measured using wall clock time. The goal of the example is to let the simulation adapt its computing cost so that the idle time between refresh data is minimal. Since the T_c is the only adaptable parameter, which can be adapted by changing the number of iterations at each time step. The T_c needs to meet the condition that $T_c + T_e$ is close to T_R, which means the total time cost between two compute actions is close to delay of the *Refresh_Flow_Data* action. Of course, this does not guarantee an optimal solution when the minimal cost for a single iteration is bigger than T_R. In those cases, other techniques are necessary to improve the adaptability of T_e and T_c. Executing the optimised asynchronous scenario, we can see the idle time that *Refresh_Flow_Data* waits for data is reduced, as shown in Fig. 5.15.

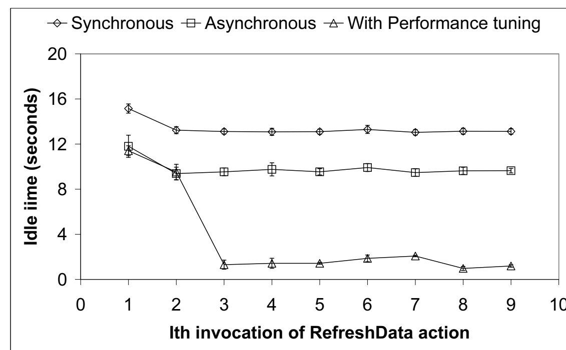


Figure 5.15: The idle time between invoking Refresh_Flow_Data *actions.*

As we discussed in the first chapter, the performance tuning of the system service includes tradeoffs between different quality attributes. In this example, reducing the computing time, on the one hand, decreases the idle time for visualisation to receive refreshed simulation results, but on the other hand, also increases the total time cost for the simulation to achieve the convergence, as shown in Fig. 5.16.

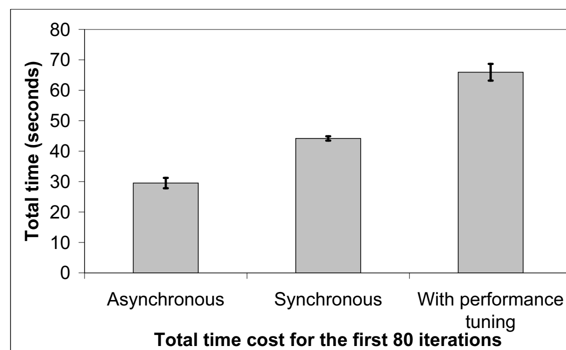


Figure 5.16: The total time for simulation to do 80 iterations computing.

5.5 Collaborative interaction in an ISS

There are a number of reasons for supporting collaborative activities in a surgical planning environment. First, the tasks of an operation involve different roles in the operation theatre, e.g. surgeon and anaesthetist; the design of a surgical plan is by nature teamwork between these roles. Second, allowing multiple users to collaboratively search for optimal operative plans can improve the efficiency for exploring possible solutions to a specific case [165, 166]. Finally, analysing data and making decisions with multiple experts can improve the soundness of a plan. In this section, we briefly discuss how the collaborative interaction is supported in the ISS-Conductor architecture.

5.5.1 Requirements

In a surgical planning environment, collaborative activities between users may vary in terms of the phases of operation planning. At the design stage, a group of users work together to prototype an experimental ISS for evaluating surgical procedures, and at run time, users collaboratively manipulate the simulation parameters and steer its computing processes. Bardram [167] classified the dynamics of collaborative activities into three layered groups: co-ordinated activities where each user focuses on his own task and passes his work to the others in a batch like paradigm, co-operative activities where a number of users share a common objective and co-operate with each other to make the achievement of the objective easier, and co-constructive activities where users can define and adapt their run-time objectives during the interaction. To support collaborative interaction, the development of an ISS needs to consider not only the conventional issues in normal CSCW (Computer Supported Co-operative Work) applications, e.g. concurrency control, data distribution, conflicts handling, and consistence maintenance [167–170], but also additional simulation specific issues. First, the shared objects being manipulated by the users are live simulations, which are not only updated by the users but also by the simulation itself; the evolution of the simulation states has to be taken into account when co-ordinating user activities. Second, an efficient distribution mechanism is needed for maintaining the performance required for timely rendering and human real-time interactions. Finally, the interaction policies between users should be customisable for being deployed in Problem Solving Environments.

During the past decade, computer mediated collaborative interactions have been studied in both the communities of CSCW and Modelling and Simulation [171–175]. The CSCW community contributed a spectrum of paradigms for supporting collaborative work between a group of users [176], and a collection of toolkits for realising them. From the perspective of implementation, three basic coupling schemes between user interfaces and the shared objects were developed: a centralised, a distributed and a hybrid one. In the centralised scheme, a co-ordinator is employed to handle the dialogues between users and to interpret the co-operation policies, e.g. in RING [177] and SCARP [178]. In some systems, the co-ordinator also manages the content that is visualised in the interfaces of the distributed users, such as in MOVE [179]. In the distributed scheme, the issues related to the co-operative interactions are handled by the front-end applications of the users, e.g. COCA [180]. In the hybrid scheme, both mechanisms are used, e.g. in Clover [181].

From a different point of view, these issues were also studied in the simulation community in the context of co-ordinating distributed simulation processes; the solutions were formulated as the standard services in the ISS supported middleware, e.g. HLA and DIS. These middlewares cannot only facilitate the interoperability between simulation processes but can also be used to support the collaborative interactions. But most of these services are low level, the high level co-ordination of user activities and of the control for collaboration polices have to be realised in the functionality of the application. A commonly used approach to separate them from the application specific

logic controls is to employ an independent interpreter for the interaction scenarios, as in the SIMULTAAN Simulation Architecture (SSA) [91].
Benefiting from the existing work, ISS-Conductor employs the basic services provided by HLA and encapsulates them as the functionality of agents. These agents provide additional services for supporting collaborative interaction. Compared to the other systems, it emphasises different features. First, an ISS-Conductor component intends to work with the existing CSCW tools; the support for general multi-user interactions, e.g. teleconferencing, are not emphasised in the architecture of ISS-Conductor. Second, it aims at wrapping legacy simulation and interactive visualisation systems, and at providing reconfigurable coupling mechanisms between them. It focuses on handling activity dependencies and for interpreting collaboration policies between the users of interactive visualisation tools, instead of on specific features e.g. WYSISWIS[‖] of the data presentation. Third, no static centralised co-ordinator is required in ISS-Conductor. Although the execution of components can also be centralised paradigm, the decision on which one acts as the co-ordinator is made by agents at run time. The support for the controlling collaborative activities is included as part of inherent function of the components.

5.5.2 Basic support

To support collaborative interactions, a number of fundamental issues have to be taken into account, such as co-ordinating activities of different users, controlling concurrent operations and distributing interaction data. The ISS-Conductor architecture has straight solutions to them.

1. *Activity co-ordination*. The user activities are co-ordinated using the scenario net. The module agent steers the users' activity by enabling and disabling the interface elements in the user interface. The Petri net based scenario net can describe basic patterns of activity dependencies, e.g. sequential, branch, merge and synchronisation.

2. *Concurrency control*. Controls for concurrent data access and update is an important issue in collaborative interaction. In ISS-Conductor, the concurrency control can be handled at both ComA and MA levels. At the ComA level, the ownership management services provided by the underlying RTI handle the modification requests on a single shared object; only the ComA holds the ownership of a shared object can update its content or delete it. At the MA level, module agents execute concurrent transitions in a scenario net (see section 3.6.1) by negotiation. The ComA level control is applied even when the MA level control is not used.

3. *Data distribution*. At run time, the shared objects are distributed between component instances through a number of routing spaces (see section 4.1.2). The

[‖] What you see is what I see.

execution states of a component are distributed between all relevant module agents in the scenario net. Shared objects are distributed among all the ComAs of the participating Actors. An MA can adapt the routing space at run time.

4. *User awareness and information exchange*. Two mechanisms allow a user to perceive the activities of the other users. First, if a user updates the content of a shared object, the changes of the shared object will be notified by the other component instances by the object management services provided by the RTI. Second, a component instance can be launched in an interactive mode, in which a user interface of the Conductor will be displayed. A user can see the visualised world model of the MA from the interface. From the interface, a user can directly chat with the others.

Apart from these, ISS-Conductor also provides two additional services for supporting collaborative interactions: user controlled scenario execution and multiple execution of a scenario template. In the coming two sections, we will explain them using examples.

5.6 Collaborative data analysis and decision making

A straight way to include a human in the execution loop of a scenario is to use the user's activity state to describe the guard conditions in the scenario net; we have seen how it works in the previous examples. However, such a mechanism has a number of shortcomings when a scenario has multiple users. First, the possible states of user activity are dependent on a specific component; they can be used in designing a scenario net only when the component is known. Second, because the scenario net does not have synchronisation control on the states of user activities; it is difficult for an MA to keep the states from different users up to date when checking them in a condition guard. ISS-Conductor provides another mechanism to do so: allowing users to explicitly express their opinions on the decision points at run time.

5.6.1 User opinions and decision points

In a scenario net, a place can be associated with a number of opinion choices and a list of roles expected to select these opinions. The opinions are used to describe the condition guards in the relation links between the place and the transitions in its post set. At run time, the information can be displayed by the GUI of the Conductor, when the place is enabled[**]; and a user can select a proper opinion via the Conductor GUI. To use the feature, the roles requested to express their opinions in a scenario have to be synchronised. It implies, first, at run time, one user can have maximally one

[**]In Chapter 3, we discussed that a place is *enabled* when it has at least one token, and the control expressions are evaluated as true.

enabled place to choose the opinions at any one time, and second, when a role has opinions displayed in its Conductor GUI, the role can not execute any other possible transitions in the scenario net. In ISS-Conductor, the state of the scenario net is synchronised between all the roles, therefore, all participant roles in the scenario have consistent state of the scenario, which means the presentation of the opinions are identical for each user. At run time, when a user selects an option from the Conductor interface, the Conductor propagates the selection to the others. When the state (number of tokens) of the place has been updated, the opinion information are cleaned.

5.6.2 Collaboratively exploring data

An easy extension to the single user scenario discussed above is to allow multiple surgeons to simultaneously explore the simulation results of the blood flow and to jointly make decisions on the current bypass. We call this scenario *Collaborative_Blood_Flow_Studying*. First, one of the users composes a bypass for an input data, and then starts a simulation of the blood flow for it. After that all users explore the results of the simulator and decide if they accept the bypass or build a new one.
We build this scenario based on the *Blood_Flow_Studying* scenario. Three roles are defined: *Surgeon_A* and *Surgeon_B* are two instances of the *C_Desktop_VRE* component, and *Blood_Simulator* is an instance of the *Flow_Simulator* component. The common data interface remains same as the *Blood_Flow_Studying_Asy* scenario. Fig. 5.17 depicts the scenario net of the interactions.
In Fig. 5.17, the Place *P8* is associated with three possible opinions: *makeNewBypass, continueSimulation* and *accept*. The execution of the scenario distinguishes the cases when both surgeons accept the simulation results, or at least one of them does not accept. At run time, the place p8 can only be enabled when both *Surgeon_A* and *Surgeon_B* have finished their action *Refresh_Flow_Data*, and no other possible transitions can be executed when p8 is enabled.

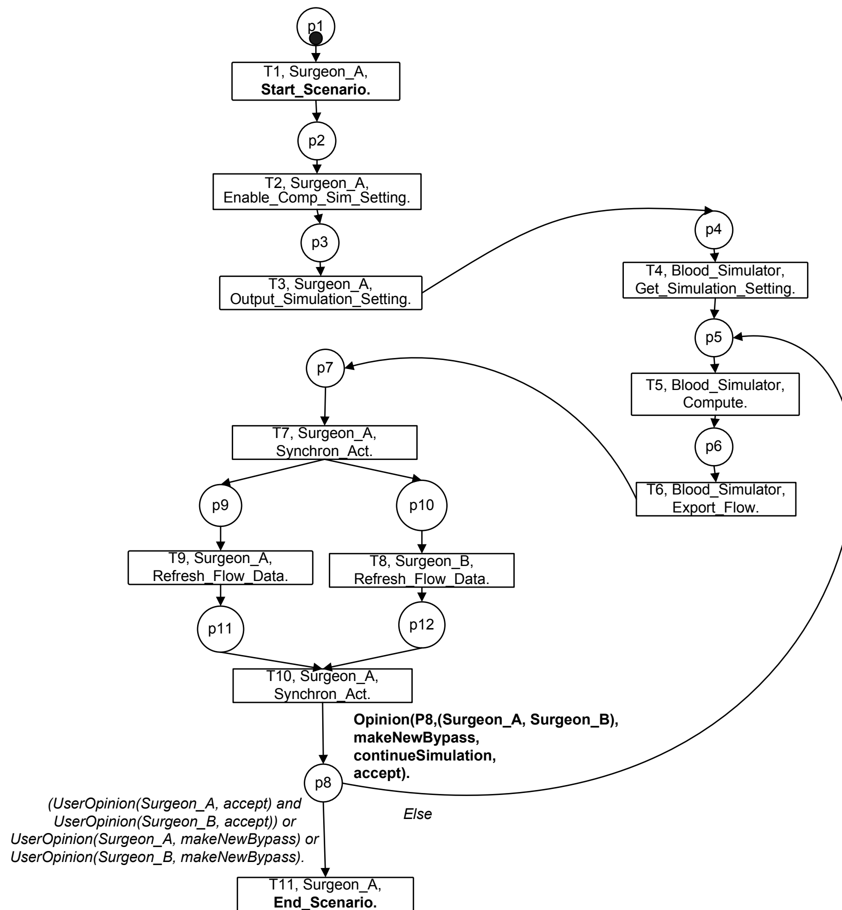


Figure 5.17: Multiple users explore the Blood_Flow *object. The term* UserOpinion *describes the opinion choice selected by a user. When both surgeons accept the simulation results, or one of them does not accept, the simulation scenario ends. The high level scenario decides whether to make a new scenario or to finish the entire story.*

5.6.3 Experimental results

Fig. 5.18 shows the run-time interface of the Conductor. The users can also use the chat interface to discuss the simulation results. When supporting multiple users to explore data, the scalability of the system is an important quality attribute of the implementation. We studied the delay for remotely updating data objects changes[††] when the number of users increases in the system. We compared the delay between a number of configurations: one, two and four surgeons in the system. The experiment was performed on the DAS II environment, Fig. 5.19 shows the average of 20 measurements.

[††]The delay is defined as the time interval from the first simulation process starts to send data until the last visualisation process finishes receiving the entire data.

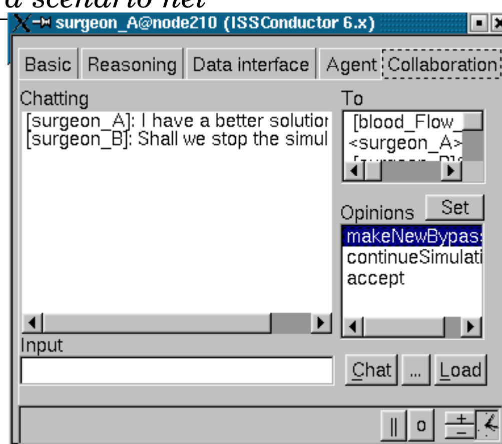


Figure 5.18: User's opinions and the chatting interface.

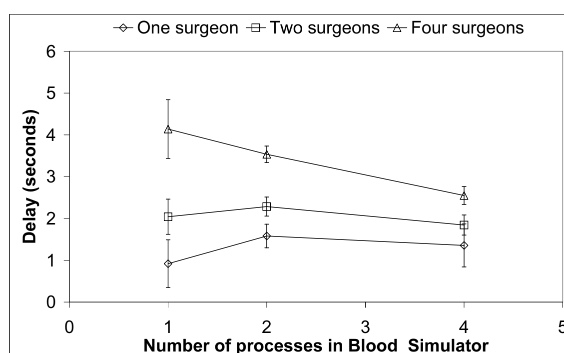


Figure 5.19: The delay for remotely updating Blood_Flow *object with one, two and four* Surgeons*. The error bars indicate the standard deviations. The size of data object is 2M bytes.*

From the measurements, we can see, when the number of surgeons increases, the time delay for all surgeons to receive the data objects from the simulation increases. This is because of the fact that the RTI distributes to the receivers one by one. But when the simulation runs in parallel, the delay for distributing data objects can be reduced, see the case of four surgeons. We knew from the previous chapter that the data objects are normally distributed to the consumers in one by one according to the order in which they joined the federation. But when more than one federate produces data objects, the parallelisation of the transmission would reduce the entire distribution delay.

5.7 Multiple instances of a scenario net

In the previous example, there is only one simulation instance to produce results for all users; this paradigm is very useful when making joint decisions on a specific simulation setting, but it can also be very inefficient when searching for an optimised configuration for a simulation model. One of the solutions is to equip each user with an independent instance of the simulation, and to allow them to interact with the simulation in parallel. In this section, we discuss how this is realised using the ISS-

Conductor architecture.

5.7.1 Scenario template and data class mapping

In ISS-Conductor, a generalised scenario net, called scenario template, is supported. A scenario template is a special scenario net, in which the role name and data classes are not concrete, and have to be instantiated before being executed. A scenario template explicitly describes the allowed non-intersected sets of roles and data classes, which can be used to instantiate the template; these sets are also called the domain of a template. A template can only be instanced by an element in the domain once, which means one role can only create and take part in one instance of a scenario template. The ComA of the role in one scenario instance can switch its data class to a different one defined in another instance. This is realised using the class mapping in ComA; we have discussed this feature in chapter 4. In HLA, when a federate subscribes a data class, all the objects of that data class are distributed to the federate within a same routing space. Declaring them as a same class but using routing spaces to control the distribution of different objects is also a solution, but that requires sophisticated controls on the region sizes of the distribution routing spaces.

At run time, a scenario template is executed in a similar way as an ordinary scenario. In the chapter 3, we mentioned, a scenario is executed when it is the top-level scenario of the story, or it is a sub scenario of another scenario net. A sub scenario is entered from a special transition defined in its higher level scenario. A role checks whether it is responsible or involved in a scenario by searching its name in the scenario net. When the scenario is a template, a role checks it from the domain of the template, since no concrete role names are defined in the transitions. Apart from it, the basic execution mechanism of the scenario is same as the normal scenarios.

During the execution, a role can switch its data object to a different one and check the situation of the other users. When a role switched its data class to a one instantiated in another template instance, the state update of its original scenario instance is paused. A scenario instance can only be ended by its creator. In the scenario template, user opinions can also be included in the description. The world model of an MA tracks the information of roles which are in other template instances.

Constructing bypasses in parallel

In a parallel paradigm, each surgeon builds a bypass and uses a separate instance of the simulator to validate it. At run time, a surgeon is allowed to switch his vision to the objects on which another surgeon is working. This scenario can be realised using the ISS-Conductor architecture. A scenario template is defined as shown in Fig. 5.20. It can be instantiated by two roles: *Surgeon_A* and *Surgeon_B*. A scenario instance ends when the creator has the opinion to accept the simulation results or wants to quit the execution.

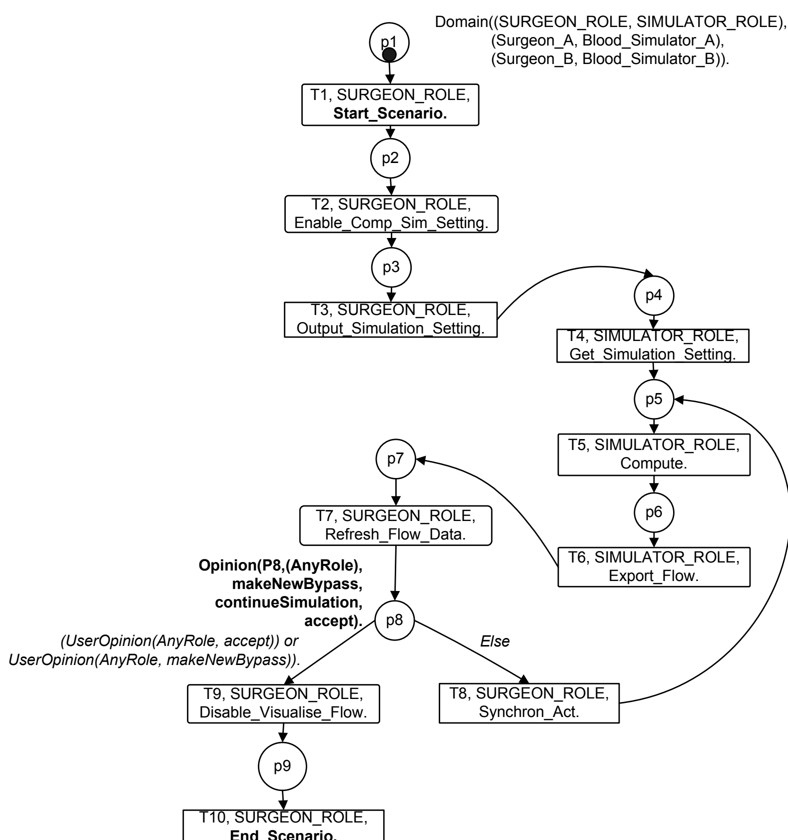


Figure 5.20: A scenario template for collaboratively searching optimal bypasses. In the template, SURGEON_ROLE *and* SIMULATOR_ROLE *are two general role names, which have to be replaced with concrete role names at run time.*

Experimental results

At run time, a user can switch the mapping of the shared class from the interface of the Conductor. Fig. 5.21 shows the GUI of switching class mapping. When a component instance switches the mapping of its shared class, its ComA will first cancel the declaration of publishing and subscribing of the original class, and then do the declaration on the class it switched to. Using the services from the underlying RTI, it will receive the latest value of the object of that class from its owner after the switch. The overhead of the switch is about 0.05 seconds.

5.8 Summarising discussion

In this chapter, we have discussed the feasibility of deploying the ISS-Conductor components to realise a pre-operative planning environment for vascular reconstruction. First, we explain the primary steps for encapsulating standalone simulation or visualisation systems as reusable and customisable components. A flow simulator and an interactive visualisation tool are used as test cases. In the system construction, we

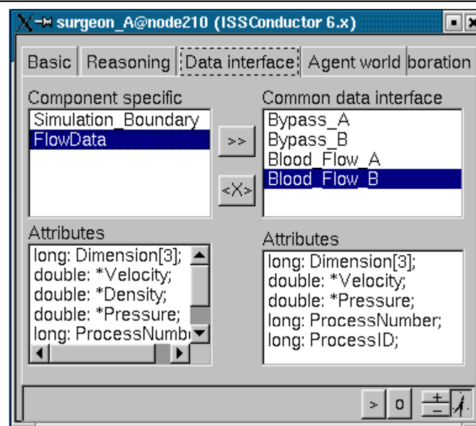


Figure 5.21: Switch shared classes between different instances of a scenario template.

highlight following issues:

1. We have demonstrated the description mechanism of *scenario nets* introduced in chapter three. In the experiments, we have shown how the control conditions can be described in a *Scenario net* using the states of component execution and user activities.

2. In the experiments, we have shown the flexibility of controlling system behaviour. The interaction scenario between components can be adapted by applying different scenario nets, and in particular the agents can take the performance constraints into account when interpreting the scenario net.

3. We have also discussed the support for collaborative interactions using ISS-Conductor. Normal flow control systems provide a minimum support for collaborative interaction: co-ordinated activities. ISS-Conductor allows human users to determine the flow execution at run time. As we have mentioned, ISS-Conductor does not aim at the WYSISWIS effects in the support of collaborative interactions.

From the experiments, we see that hiding the low-level details of interconnection improves the efficiency for prototyping an interactive simulation system. The experimental results show that ISS-Conductor has acceptable performance for human interaction. Since the main goal of ISS-Conductor is the scenario description and execution instead of the system performance, we have not discussed much about high-level optimisation on system performance.

5.9 Conclusions

Simulation aided surgical planning is an important test bed for interactive simulation systems. In order to allow such systems to be deployed in real situations, the cost of system development must be strongly reduced. Component-based engineering

technologies are emerging as a promising solution. From the experiments, we can say that ISS-Conductor is a suitable architecture for rapidly prototyping such systems. It leads following conclusions:

1. Using the ISS-Conductor architecture, legacy simulation and visualisation programs can be encapsulated as reusable components, and can be used for rapidly prototyping interactive simulation systems.
2. Separating application logic from the component functionality improves the flexibility of controlling the overall system behaviour. The agent framework in the ISS-Conductor architecture encapsulates the control intelligence for interaction constraints between components and provides an explicit layer for adapting system behaviour.
3. Dynamically tuning system performance is a necessary optimisation mechanism to improve the service quality of a run-time system. It complements the activity based scenario control with concerns of quality of service.
4. The ISS-Conductor architecture supports collaborative interactions at the scenario level.

Chapter 6

Towards an intelligent planning environment for interactive simulations

In the previous chapters, we discussed the architecture of ISS-Conductor and its utilisation in constructing interactive simulation systems. The layered integration mechanism in ISS-Conductor improves the flexibility of controlling the application logic of an ISS. Yet, the difficulties of describing Petri net based scenario nets may also hamper the introduction of ISS-Conductor in Problem Solving Environments. In this chapter, we discuss an approach to this problem in a proposed environment called Interactive Simulation System Studio*.

6.1 Introduction

Problem Solving Environments integrate computing technologies and provide an abstract environment for scientists to do research on various problem domains. Since the 1980s, Problem Solving Environments have become an important subject in the community of High Performance Computing [182–185]. Depending on the target domain of the system and the freedom that the scientists are allowed to customise the system interaction, a PSE may have different guises, e.g. an Interactive Simulation System with customisable configurations [186], or a library of solvers and its necessary user interface as in [16]. But at an abstract level one can always distinguish three main functional subsystems in a PSE: an environment for analysing problems and designing experiments, a collection of necessary software resources for building experiments and an environment for executing the experiments.
PSEs play a key role in the emergence of computer simulations, and in particular interactive simulations, as an important experimental paradigm for problems that

*Parts of this chapter have been published in *Z. Zhao, G.D. van Albada, A. Tirado-Ramos, K.Z. Zajac and P.M.A. Sloot*. "ISS-Studio: a prototype for a user-friendly tool for designing interactive experiments in Problem Solving Environments", in the proceedings of ICCS 2003, Melbourne, Australia and St. Petersburg, Russia, Part I, in series Lecture Notes in Computer Science, June, 2003.

are difficult to solve using conventional methodologies like experiments using normal lab instruments. As we discussed in the earlier chapters, the complexity of implementing an ISS lies in three main aspects: developing *valid* simulation or visualisation kernels, coupling distributed modules of the system, and controlling their run-time activities. Employing the simulation or visualisation kernels from legacy systems can reduce both the risks and the costs of the development of an ISS. However, the customised integration mechanism resulting from such construction paradigm introduces a strong dependency between the constituent system modules and hinders the further deployment of the system in PSEs. One of the solutions is to use software component technologies: industrial components, e.g. Java beans and DCOM, and scientific computing components, e.g. CCA, have been used to encapsulate the simulation and visualisation systems and to facilitate the interoperability between them [16, 89, 90, 184]. Most of the available architectures provide a description mechanism to specify the functionality of the components, e.g. the SOM (Simulation Object Model) in HLA and the SIDL (Scientific Interface Description Language) in the CCA, and an integration mechanism for assembling the components and for realising their run-time binding. In those architectures, the interface specifications are basically used to promote the interoperability between components; an explicit layer for controlling overall interactions is not defined. Using these architectures, complex activity constraints, e.g. multi-user interactive simulation, are often difficult to describe at the flow control level. Therefore low level component programming is needed, which still hampers the further introduction of ISSs in PSEs. Hiding these low level assembling and programming details from a scientist and allowing him to plan an experiment at a high level is desirable. Since the planning procedure will be partly automated by the system, we call this the *intelligent planning of interactive simulations*.

The intelligent planning is basically approached by mechanisms which support automatic (or semi-automatic) selection of components and derivation of the coupling details between them. The research on this subject received a great deal of attention after significant progress was achieved on the reusability and interoperability of the simulation components [187–189]. An efficient mechanism for selecting software components has been considered as a necessary step to approach the intelligent planning. A number of technologies were reviewed in [187], e.g. based on key words, facets, signatures, behaviour and semantics. One of the conclusions drawn from the paper is that semantic level component matching is essential to improve the searching efficiency. A number of researchers studied the feasibility of automating the compatibility check between the Simulation Object Model (SOM) and the Federation Object Model (FOM) of an HLA application, but most of the matching mechanisms are limited to the syntactical level, e.g. in [188]. Using predefined templates, e.g. Process Flow Templates [15], is a straightforward way to facilitate the composition of interactions between components. But the templates are mostly composed manually by domain specialists. The burgeoning applications of Service Oriented Computing [190] paradigms are an important force to push the research on automatic flow composition. One of the motivations is to compose the flow between intermediate services and to provide a transparent binding interface for the service requester. A number of

researchers have studied this problem in both architectures of web services and Grid services [189, 191, 192]. The basic idea is to distinguish the dependencies between the services according to their pre and post conditions on data, and describe them using a workflow description language.
The ISS-Conductor architecture provides solutions for encapsulating legacy simulation and visualisation tools, and for orchestrating their activities at run time. A Petri net-based control mechanism for component activities supports the description of sophisticated interactive scenarios. Automatically planning of ISS-Conductor based experiments exhibits a number of differences from the other related work. First, the capability descriptions of ISS-Conductor components are based on state machines; they provide extended information for the simulation object models, and thus it becomes feasible to include more sophisticated matching mechanisms than in [188]. Second, the execution of an ISS-Conductor system is based on HLA, but the interaction scenarios between the components are based on Petri nets; they provide semantics to complement the object model based composition with the activity constraints between components. One of the aims in this chapter is to study the feasibility of composing scenarios which support human-in-the-loop computing.
In this chapter, Interactive Simulation System Studio (ISS-Studio), a framework for deploying ISS-Conductor components in constructing interactive simulation based experiments will be proposed. First, we give an overview of ISS-Studio and enumerate its desired functionality. ISS-Studio is proposed specifically for the ISS-Conductor compliant software resources. It intends to work with existing generic PSE frameworks to enhance their services for supporting interactive simulation based experiments. We will discuss this issue using an example of Grid-based Virtual Laboratory Amsterdam (VLAM-G) [15, 19], a general PSE framework developed at UvA. After that we discuss the basic procedures to automate the story composition for an ISS-Conductor based system. Finally, some experiments and earlier results will be presented.

6.2 A global picture

The main goals to propose ISS-Studio is to facilitate the development of ISS-Conductor based components and to simplify the construction of interactive simulation systems. In this section, we will first describe the desired functionality of ISS-Studio and then discuss the design requirements for them.

6.2.1 Proposed functional subsystems

From the lifecycle of developing ISS-Conductor based components and interaction stories for the integrated systems, the functionality of ISS-Studio is grouped into four subsystems: component management, knowledge management, experiment planning and run-time experiment management.

Component management

Incorporating existing standalone tools which are designed for a specific problem as reusable and customisable solvers for a spectrum of other problems [92] is an important way to enrich the software resource of a PSE. The first subsystem will aid component developers to incorporate legacy simulation or interactive visualisation programs into the ISS-Conductor architecture. The component management subsystem provides tools for component developers to construct and maintain components, e.g. to define a component capability, to develop code and to debug. The component products, including the capability specification, the source, the documentation and the binary are stored in repositories with version control. Services for retrieving and updating components from the repositories are also provided.

Knowledge management

An efficient reuse of the software components depends not only on the nature of the components but also on the mechanisms for searching and retrieving them from the repositories where they are stored. As we mentioned, conventional search techniques do not capture the run-time semantics of the components. In ISS-Conductor components, the actions are complemented with pre and post conditions: the requirements and influences on the data objects, but they do not guarantee that the retrieved actions provide the semantics that the component searching process needs because of the possible diverse meaning of the actions and data classes. One of the solutions is to synchronise the meaning of the vocabularies used in different repositories using a knowledge-based backbone; the concepts used for describing software resources, e.g. components and experiments, are associated with certain ontologies. A knowledge management subsystem is proposed for this function.

Experiment planning

The third subsystem is to plan ISS based experiments. It intends to aid a scientist to develop an interactive simulation based experiment at each phase of the lifecycle. The subsystem needs to provide a user-friendly environment and supports intelligent planning of the experiments. In the next section, we will come back to this point.

Execution management

An interactive experiment is executed on computing resources, e.g. supercomputers, clusters or high performance virtual reality environments. The fourth subsystem processes the resource requirements of an experiment and generates suitable job description for different types of computation resources. The execution management subsystem also provides an interface to interact with tools for execution monitoring and job migration.

6.2.2 Design requirements

The system must meet a number of requirements. The first one is the user-centred design; the system needs to consider different types of users, e.g. component developers and scientists, and their special requirements on the system interactions. Second, integrating commercial off-the-shelf (COTS) tools into the system is another important requirement; using mature COTS tools e.g. for supporting UML or XML, avoids unnecessary rebuilding of similar utilities. The third requirement is the portability of the implementation. The constituent tools of the system are likely to be distributed. Thus, a portable framework to glue these assets is needed, and in addition, diverse interfaces to access these tools can also improve the usability of the environment. The realisation of the system needs to benefit the existing platforms, such as the management of distributed resources in Grid environments. Finally, the feasibility to integrate with existing generic PSE frameworks also has to be taken into account. Realising special purpose PSEs using a generic framework has emerged as an important development paradigm [16, 19]. Services provided by these frameworks, e.g. for managing resources and run-time information, can simplify the development of ISS-Studio.
In the next two sections, we will first discuss how available Grid middlewares can contribute to the development, and then use VLAM-G as an example to discuss the feasibility to deploy ISS-Studio in existing PSE frameworks.

6.2.3 ISS-Studio and Grid environments

A core idea of Grid environments is to organise heterogeneous resources, e.g. computing elements, storage devices and software components, and share them among a group of trusted users (Virtual Organisations)† [194, 195]. Resource management is a central component in a Grid environment, it provides services for describing and discovering resources, for scheduling and monitoring them at run time, and for fault tolerance and security control. A number of resource management systems have been developed, e.g. Condor [61], Globus toolkit [196] for computing resources, European Data Grid [197] for data resources, and PUNCH [198] for services-based resources. For instance, in the Globus toolkit, resources are described using an extensible resource specification language (RSL), the resource requests are handled by resource brokers and processed through the information service provided by meta-computing directory services (MDS). In the Globus toolkit, job schedules are organised in a distributed paradigm.
The rich set of protocols defined in available Grid environments provides a suitable infrastructure for realising ISS-Studio, e.g. for component storing and discovery, and for execution monitoring. There is also a large research body on flow control in Grid environments [16,137,138]; most of them are based the Grid Service architecture and describing the flow using data or task based dependencies, e.g. in GridAnt [138] and

†Grid environments are also classified as: computational, data and service Grids according to the type of resources being managed and shared, e.g. computing elements, storage and software [193].

Taverna [137]. Compared to them, ISS-Conductor allows more sophisticated controls: the states of human activity and the components execution are allowed to control the flow branches, but the framework is currently based on HLA.

6.2.4 In the context of a PSE framework

VLAM-G is a generic PSE framework, which provides hierarchical solutions to manage different levels of resources, and encapsulates them as services in a middleware. On top of the middleware, domain specific PSEs are supported. The middleware allows users to work simultaneously and collaboratively at different levels of the framework, e.g. as scientists, domain experts, tool developers and ICT[‡] developers. It also integrates the information management services with the lifecycle of a scientific experiment [21]. An experiment is modelled using *physical entities* which are the instruments to be used, *activities* to be performed by the scientists, and *data elements* which are the input/output of the activities. An experiment is described as the flow between these elements; in order to simplify the construction of an experiment, templates of the flow are abstracted as Process Data Flow templates. A database infrastructure is employed to manage both the static and run-time information.

Compared to VLAM-G, ISS-Studio uses the term *experiments* in a much narrower sense. In ISS-Studio, experiments only refer to the interactive simulation based paradigm, and they can be included as part of a VLAM-G experiment. ISS-Studio focuses on the mechanisms that can facilitate the composition of ISS-Conductor based experiments. In the context of VLAM-G, ISS-Studio can be viewed as an upper level PSE, where the subsystems can benefit from the services provided by VLAM-G middleware, e.g. for managing resources and experiment information.

In this chapter we will not discuss the detailed issues on using the Grid services to realise ISS-Studio, but instead we focus on the intelligent planning of ISS-Conductor based interaction scenarios.

6.3 Intelligent planning of ISS-Conductor based interactive simulations

In this section, we will focus on the experiment planning subsystem and discuss the feasibility of intelligent planning of ISS-Conductor based interactive simulations. The goal of the subsystem is to allow a scientist to plan his interactive experiments from the level of the problem domain instead of the details of scenario nets composition. We will briefly discuss the phases: requirement description, component discovery, story making and execution script generation.

[‡]Information and communication technologies.

6.3.1 Describing experiment requirements

The first phase aims to describe the requirements of the experiment. The description will be the input to the experiment planning environment. It provides information for the subsystem to determine the suitable components for the experiment and to distinguish the interaction constraints between the components in a story. The goal of employing interactive simulation in a scientific experiment is to use simulation solvers to compute data properties of a model, and to allow the scientist to study them by manipulating part of the data at run time. Therefore, we argue that the experiment description should at least contain three main elements: data, activity and the quality requirements.

1. *Data*. A scientist needs to specify the data for an experiment. It describes not only the raw data that the user has but also the data he expects during the experiment.

2. *Activities* indicate the action that the user will perform on the data. Some activities also indicate the transformations between data or causal relations between the data.

3. *Quality requirements* on data and activities describe the performance constraints of the experiment.

This model has a number of advantages. First, the activity flow of an experiment by nature is a sequence of operations on the simulation data. Although the implementation information of the simulation and visualisation kernels is not explicitly modelled, they can be included in the description as the quality requirements on the data or the activity. Second, mature software modelling techniques, e.g. data flow and control flow, can be directly used to describe the experiment. The description can be intuitively represented using graphical primitives. Fig. 6.1 shows an example of describing a bypass validation experiment. Finally, the description can be parsed and described using a logic language, e.g. first order logic, which can be parsed and reasoned on by agents for further searching and composition.

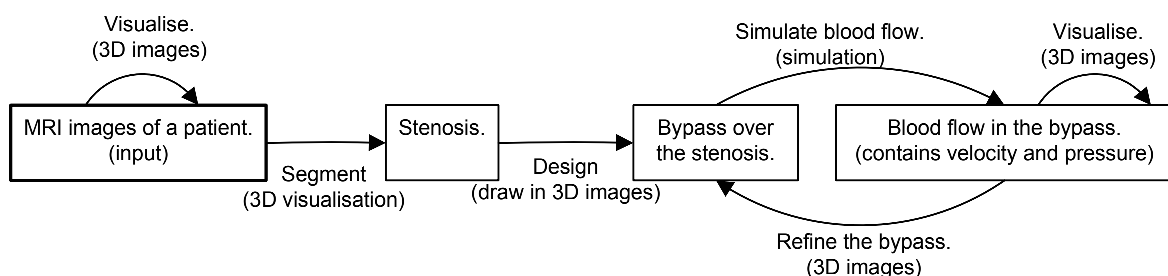


Figure 6.1: A graphical representation of the experiment requirement for a bypass-validation experiment.

6.3.2 Component searching

The actual planning procedure starts when the user provides an experiment description. Component searching is the first step, it finds a set of suitable components which can 1) produce all required data, 2) support all activities on them and 3) provide services with the required quality. In ISS-Conductor, the capability of a component is described based on a finite state machine model, in which data classes, activities, and the dependencies between the data and activities are explicitly described. It provides search agents basic information for match checking. Table 6.1 shows the details.

Table 6.1: Information source for evaluating experiment requirements.

Component capability	Experiment requirement
Data classes (shared and internal)	Data
Activity	Activities
Pre-condition and post-condition of Activities	dependencies between data
Quality attributes	Quality requirements

As we mentioned above that the syntactical level matching does not guarantee the semantic level consistency between components and the requirements because the terminology used in the description of component capability and in the experiment requirement might have different meanings. One of the solutions is to synchronise the meaning of these concepts using a consistently defined Ontology.
Originally, the term of ontology refers to a philosophical discipline for dealing with the *nature and the organisation of being* [199]. Recently, it is used in computer science as a term for describing the semantic relations between the symbolic representations and the actual meaning of concepts; it normally consists of a vocabulary and a set of explicit assumptions regarding the intended meaning of the vocabulary [200]. The assumptions are represented using logic theories, e.g. first order logic or description logic. Based on the level of generality, different types of ontology are often distinguished as a hierarchical scheme, as shown in Fig. 6.2, [201]. According to the classifications, we define four groups of ontologies. The ontologies in the top-level group describe the most general concepts, e.g. ISS-Conductor components, component instances and interactive experiments. The ontologies in a domain group describe the concepts of different domains in software resources, which cover the terminology used in defining data object models in components and interaction stories. The ontologies in a task group describe the concepts of activities, services and their quality attributes, which are related to software resources. Finally the ontologies in an application group describe the concepts bound to specific applications. Taking the example in the previous chapter, the ontologies in the domain group describe the concepts in defining data object models in components and interaction stories e.g. fluid flow, blood, flow velocity and pressure. The ontologies in the task group include the concepts for describing component activities, e.g. flow simulation, visualising MRI images and designing bypass.

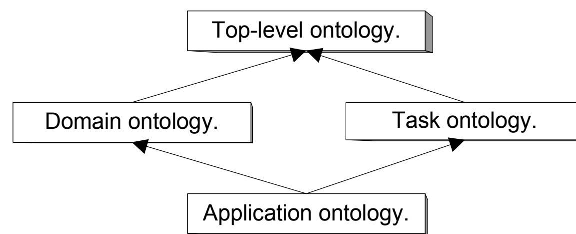


Figure 6.2: Different types of ontologies.

Using an ontology language, like OWL [202], concepts are described as *classes*, which can have subclasses and be a subclass of another class. A class can have a number of attributes, called *properties* or *roles*; the value restrictions on the properties are called *facets*. The instances of a classes are called *individuals* of the class. An ontology together with a set of individuals of classes constitutes a *knowledge base*. Fig. 6.3 shows a screen snapshot of top level ontologies (edited using Protégé [203]). The terminology used in the resource descriptions, e.g. component actions, states and data classes, are mapped as individuals or subclasses of the classes in the ontologies. A resource description can be associated with more than one ontology.

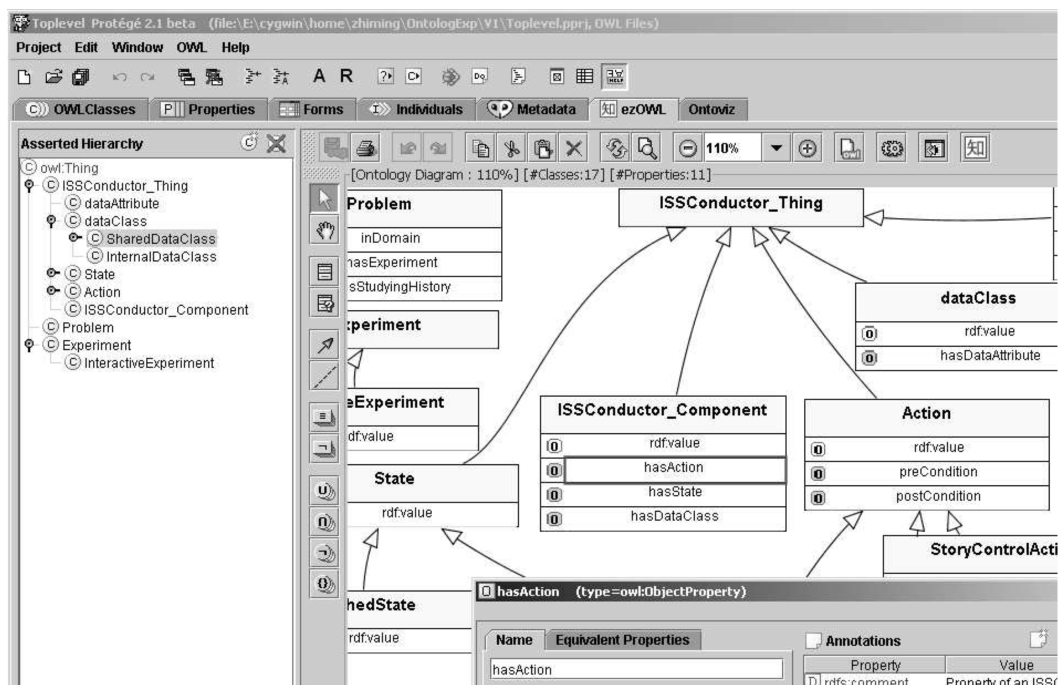


Figure 6.3: Developing Ontologies using Protégé.

In the searching procedure, the similarity between the concepts is first checked using ontology reasoning algorithms. Only the components which have equal or similar meaning of the concepts as the experiment description will be checked for the further matching.

6.3.3 Story generation

A story can be generated when the components have been found from the component repositories. It is a procedure to assemble these components and make a story for them to work together. It has a number of detailed steps.

1. The first step is to substitute the data and activities in the description of the experiment requirement using the components found. During the substitution, the roles of component instances and the common data interfaces between different roles will be defined.

2. Second, according to the condition and dependencies between activities, an intermediate activity diagram will be derived. In the activity diagram, the activities will be associated with specific role. In this step, a user can refine the control conditions between components in the loop.

3. Third, the control patterns of the activity diagram will be mapped onto Petri net. The activities and its responsible roles will be mapped onto transitions, and the conditions between activities are mapped onto places.

4. Finally, the Petri net is output as a story.

6.3.4 Generating execution scripts

The final step is to map a story onto the job description scripts of computing elements. The story contains information about the computing requirements of each of the components, e.g. requirements on the parallelisation libraries and hardware platforms, which can be used to generate a job description script using the demanded syntax provided by the description language of the computing elements.

6.4 Prototype and preliminary results

The implementation of ISS-Studio is still ongoing. In this section, we will describe the basic techniques that are used in the prototype and discuss some experimental results.

6.4.1 A multi-agent based experiment planning environment

ISS-Studio will be a distributed environment; when there are a large collection of component repositories, using multiple agents can improve the efficiency for component searching. A multi-agent environment is proposed for the experiment planning subsystem. An experiment manager agent (EMA) provides a graphical interface for users to describe the experiment requirements, and co-ordinates search agents to find

suitable components for the experiment. Component search agents (CSA) scan component repositories and search components according to the requirements sent by the EMA. Finally the EMA also does the story making and execution script generation.
The agents are prototyped using the JADE, a Java-based agent development framework for the FIPA standard [204]. In JADE, agents communicate using an Agent Communication Language (ACL) and are managed by an agent container at each host. The JADE framework provides services for managing the lifecycle of agents including cloning and migrating them between hosts. In an agent, the reasoning kernel of the agents is realised using Prolog. The ontology-reasoning module is realised using Racer [205] which can be shared by different agents. Fig. 6.4 shows its basic agent architecture.

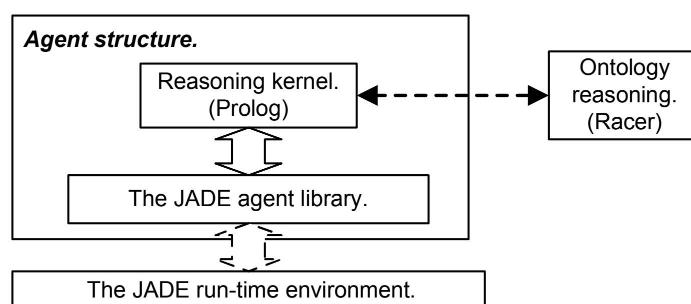


Figure 6.4: The basic architecture of an agent.

6.4.2 Experimental results

Experiments for testing the feasibility of integrating the JADE framework, Racer and the Prolog reasoning kernel have been performed. Because of the powerful support for network-based programming, SWI Prolog [206] is used in the prototype. The JADE framework is based on Java, and both SWI Prolog and Racer have a Java-based interface. The descriptions of component capabilities and the experiment requirements are parsed as Prolog terms; the Ontologies are in OWL and are processed by Racer. The Prolog reasoning kernel communicates with the Racer server via sockets. The JADE framework handles inter-agent communication. By gluing them using Java, the basic control between the functional components of an agent can be realised.
The GUI of component management and experiment planning subsystems have been prototyped using a Java based graphical library, JGraph [207]. Fig. 6.5 shows a screen snapshot. The GUI allows a user to directly describe the activity-transition graph (see Section 3.2.1) of the component and export as the ISS-Conductor required format. Fig. 6.6 shows a screen snapshot, which shows the experiment requirement described in Fig. 6.1: the rectangles and ellipses are respectively represent data and activities. The requirements are described using a set of triples, which are transformed into Prolog lists. From the *planning* menu, a user can start a CSA to discover the suitable components.

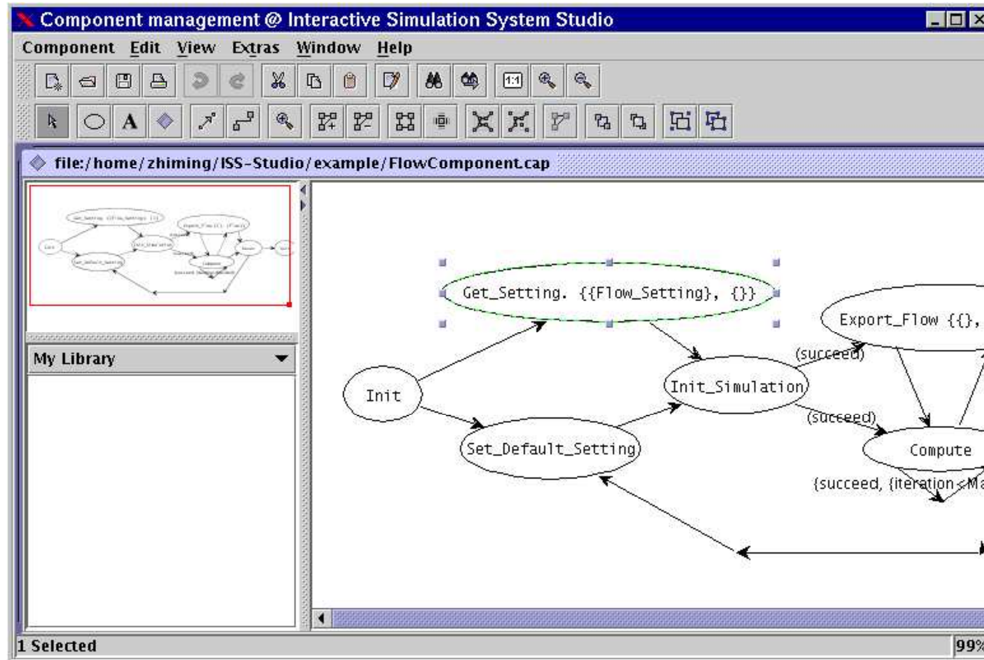


Figure 6.5: A snapshot of the component management subsystem.

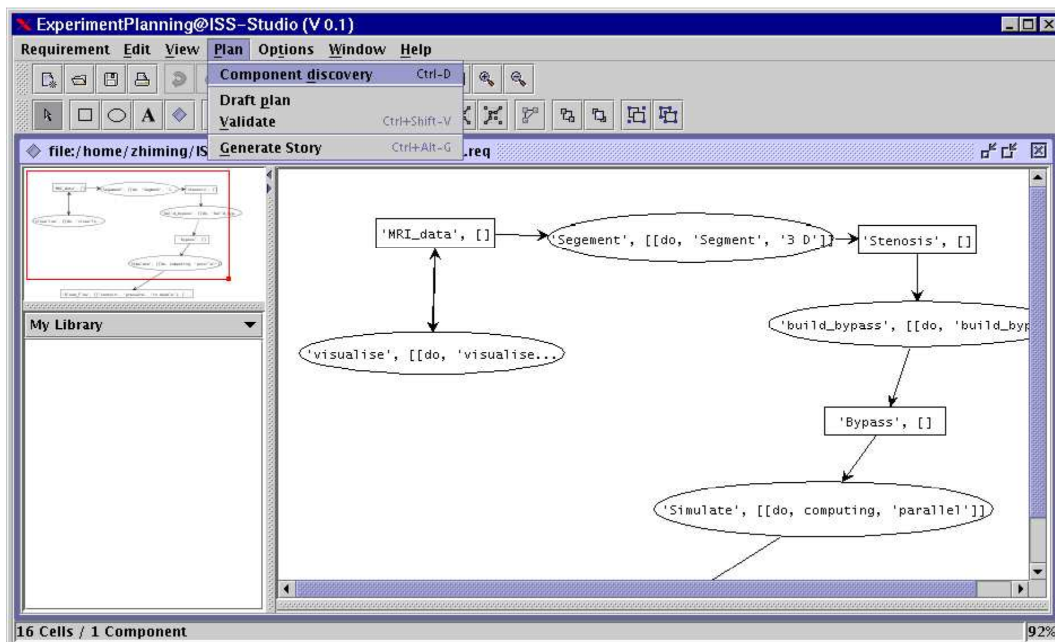


Figure 6.6: A snapshot of the experiment planning subsystem.

A reasoning kernel for a CSA has been prototyped. The prototype is able to find components from a given collection using the matching rules discussed above: the set of components can perform all the activities and process the data requested in the

requirement description. Since we do not have a large collection of ISS-Conductor components yet, the components (in total five) we discussed in the previous chapters are used as the basic collection. For the experiment purpose, we replicated them and created a number of dummy components; in total the component collection contains 20 samples. The capabilities of these components are parsed into 700 Prolog terms. In the experiment, the prototype can find components for each required activity, except the first two, visualise and segment MRI images (see Fig. 6.1). To get the feeling on search complexity, we run the experiment with different number of requested activities. Fig. 6.7 shows the measurements for two situations: all the requested activities and none of them can be found from the collection. For each requirement, the search procedure stops when it finds the first suitable component. When there is no component qualified for an activity, the search procedure takes more time, since it has to scan the entire collection of components, which indicates the upper bound of the searching time cost. Although the number of components is relative small, we can observe that, the search cost increase linearly with the number of requested activities in the requirement description.

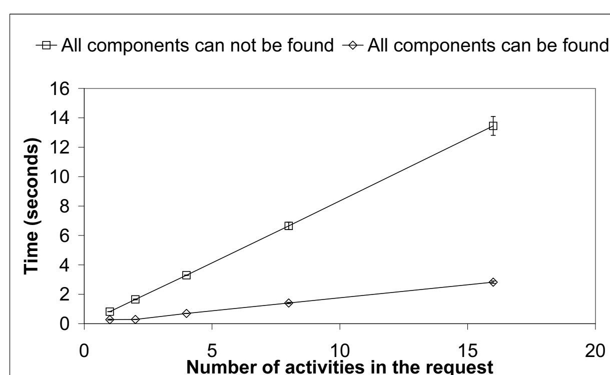


Figure 6.7: Searching different number of activities from the component collection.

6.5 Discussion and conclusions

In this chapter, we have reviewed the background of Problem Solving Environments and their role in modern scientific research, and then discussed the feasibility of developing a framework for deploying ISS-Conductor based components to prototype interactive experiments. ISS-Studio introduces a number of novel ideas in its design. An agent based support environment performs resource discovery and the interaction scenario composition, which intends to automate the procedures of experiment planning. In the other PSEs [114, 208, 209], the human guided assembly of components and interaction descriptions are still the principal development activity. Ontology based concept checking is highlighted in the component discovery.

This leads to the following conclusions.

1. Component technology is suitable to encapsulate the functionality of software resources and to integrate them in a layered paradigm.

2. A semantic level searching mechanism is a key issue to automate the utilisation of component resources. Using a knowledge based backbone to synchronise the meaning of the concepts in the component specification enhances the traditional searching mechanisms.

3. An agent-based framework is suitable for realising the experiment design environment for problem solving; employing agents to search for components and to plan experiments distributes the computation onto available resources and helps to achieve a better resource utilisation.

4. The rich set of services provided by the Grid environment constitute a suitable infrastructure for ISS-Studio to manage the component resources. We have not explicitly discussed the integration of ISS-Studio with Grid middleware. In our opinion the Java based platform and the XML based information description of ISS-Studio make the realisation possible.

Chapter 7

Summary and discussion

7.1 Summary

In scientific research, interactive simulations play an increasingly important role; compared to conventional simulation systems, they allow parameter spaces to be explored more efficiently and computing processes to be controlled more accurately. In this way they also help to optimise the resource utilisation for both computing and storage, and to improve the efficiency of communication. However, the development and integration of the simulation and visualisation kernels is expensive. Traditional development methods often result in systems that have a limited adaptability due to the strong dependency between the system functionality and application specific logic control. The high costs and limited flexibility hamper the introduction of such systems. The available software architectures and middlewares for simulation systems mostly focus on the interoperability between system components, but not explicitly on the support for rapidly prototyping ISSs and for flexibly controlling the system behaviour.

In this thesis, we state that *the separation of application specific logic control from system functionality constitutes a crucial step in an improved development process for ISS*. We developed an agent-based component architecture, called Interactive Simulation System Conductor, implementing such a separation. It encapsulates the control intelligence for the integrated system as different roles of agents, and allows to employ these agents to realise a layered interconnection between components. The proof of concept implementation is based on High Level Architecture, a dedicated middleware for distributed simulation systems.

In the second chapter, we introduce the basic architecture of ISS-Conductor. Encapsulating the computing kernels and support structures of the principal modules of an ISS, such as simulation, visualisation and interaction, it allows scientists to assemble simulation experiments at a high level without concerning themselves with the low level details of integration. In the architecture, Communication Agents realise the basic interoperability between components; Module Agents orchestrate the

run-time system behaviour. The Run-Time Infrastructure of HLA is employed as the software bus for binding the distributed components.

In the third chapter, we discuss the functional design of ISS-Conductor. In an ISS-Conductor component, the MA controls the component behaviour using a reasoning kernel; it has knowledge about both the capabilities of the components and the application specific interaction constraints in its knowledge base. The component functionality is modelled as a finite state machine (*capability*), which can be programmed with the other components using a Petri net based mechanism (*scenario net*). At run time, MAs collaboratively interpret the interaction constraints and realise the control of the overall system behaviour.

The implementation details and performance characteristics of ISS-Conductor are discussed in chapter four. The ComA interfaces with the basic RTI services for data sharing and message passing. The reasoning kernel of the MA is realised using Prolog. We measure the performance of the implementation, and conclude that the Communication Agents add acceptable overheads to the RTI. ISS-Conductor based applications can reach a network utilisation comparable to that of pure TCP sockets for large size data objects. The logic control in the reasoning kernel also adds a small overhead.

In the fifth chapter, we employ the ISS-Conductor architecture to prototype an interactive simulation environment for planning vascular operations. We discuss the detailed procedures for developing an ISS-Conductor component and for assembling the components into an interactive simulation system. In various test cases, we discuss the scenario level activity control, automatic performance tuning in a run-time system and collaborative interaction support. The experimental results show that ISS-Conductor only adds a small overhead to the legacy computing kernels.

In the sixth chapter, we discuss the feasibility of including ISS-Conductor compliant components as software resources in a Problem Solving Environment. One of the crucial issues we investigate is the automatic composition of an interactive simulation based experiment. ISS-Studio, a Java based environment has been prototyped in this chapter.

7.2 Conclusions and discussion

In the thesis, we describe the layered architecture of ISS-Conductor and discussed its implementation details. We demonstrated its capability to realise the separation between the ISS functionality and the application logic control, and therefore to fulfil the mission that we set out for in the beginning of this thesis: providing a layered framework for rapid prototyping of interactive simulation systems. A proof of concept implementation is based on HLA. In this final chapter of the thesis, we will not repeat

the conclusions drawn in the previous chapters, instead we discuss the architecture from the perspective of the underlying middleware.

The continuous growth of computing power and the design of novel middleware offer new possibilities for realising component integration and interoperability, but they also introduce incompatibility between legacy implementations and the new platforms. The heterogeneity of the computing platforms will be permanent. The solution used in the software industry is to establish a steering group, e.g. the OMG [210], from the main members of the community to formulate platform independent architectures for the development. However, the software development in academia can not always benefit from such mechanism. One of the reasons is that the effort in developing novel simulation and visualisation systems mostly lies on studying the domain problem rather than the engineering itself. The problem for integrating legacy simulation and visualisation systems is thus going to remain and there will be no single solution to all situations. In this thesis, we studied this problem from the perspective of rapidly prototyping ISSs, and contributed an agent based architecture to couple simulation and visualisation systems.

Considering a component based ISS as an analogue of an electronic circuit system, an ISS-Conductor component is more like an *intelligent card* than a *silicon chip*; it encapsulates the necessary intelligence for integrating a number of functional units together with the control for their run-time behaviour. There are a number of reasons for us to choose this vision. First, one of the missions for developing ISS-Conductor is to support the rapid prototyping of interactive simulation system. Using ISS-Conductor components, an ISS can be defined at the level of system interactions, which is easy to map onto the description of the domain problem. The assembly of small size units is considered to be a task for the component developer rather than for the research scientists. Second, this mechanism also has weak requirements on the underlying framework; ISS-Conductor only requires necessary services for distributing data objects and timestamp-ordered messages. The ComAs in the architecture localise the dependencies on the software bus and a component can be equipped with different implementations of the ComA to achieve the portability. Third, the run-time activities are controlled by the components autonomously instead of by a separate co-ordinator. It can have flexible paradigms for controlling the execution, both distributed and centralised.

HLA is a suitable architecture for distributed simulations, but it will not be the final solution. In this thesis, we did not discuss the issues on integrating the ISS-Conductor architecture with the Grid platforms. Actually, work along this line has been initiated in a European project, CrossGrid. In [211], we proposed a layered approach to realise the run-time binding between HLA compliant components on Grid environments. At the RTI level, the communication endpoint of the RTI execution is wrapped as a service, which can be discovered by an HLA application to create a federation. At the federation level, the basic services for managing HLA applications remain, only the data distribution between federates is handled using the

Grid enabled facilities, such as Grid FTP. At the federate level, the services originally provided by the RTI will be wrapped as Grid services. The contribution from this work can be applied to ISS-Conductor, since from the implementation point of view ISS-Conductor components are HLA compliant federates. The migration will not change the main design of ISS-Conductor but only the implementation of the ComAs. Further progress is discussed in [212, 213].

7.3 Future work

Based on the results of this thesis, a number of future research lines can be envisaged:

1. A lesson learned from VLAM-G project is that scientists will not choose a novel architecture simply because it looks beautiful unless it can work with the existing ones and provide exciting new features [214]. An important future work is to study the mapping schemes between existing architectures and ISS-Conductor so that legacy architectures can get benefit of the layered integration from ISS-Conductor in an easy and efficient way.

2. Looking at the underlying middleware, new integration paradigms provided in Grid environments, such as service-oriented integration, will be considered as new basis for ISS-Conductor.

3. The work described in chapter 6 is still on going. Including ISS components as software resources and sharing them among different organisations and projects is one of the core paradigms of combining ISSs and Grid environments. The research issues for semantic based discovery and intelligent planning will form another line of future work.

List of Figures

1.1 Functional components and the data flow in a simulator. The time stepping routines define the actual behaviour of the simulator, and the control routines define the experiment performed on the simulation. 2

1.2 Computing power increases in the past decade. The figure shows the fastest (N=1) and the slowest (N=500) computer in the list, and the total performance of all computers in the list. The original information is from the website http://www.top500.org. 3

1.3 A general data flow diagram of visualisation systems. The procedures in a visualisation pipeline constitute the core of the system. The generation of intuitive primitives for rendering, and the user interaction that controls the execution of pipeline can be performed in different machines. 4

1.4 A basic configuration of ISSs. Solid lines depict the simulation loop, and the dash lines depict the visualisation loop. 6

1.5 Basic components in SPLICE: application processes, each with an agent and a local data store. 9

1.6 Distributed federates and the Runtime infrastructure (RTI). 10

1.7 Summary of available architectures. 19

2.1 Basic architecture of an ISS-Conductor component. 25

2.2 A simple agent kernel. 26

2.3 An Actor and its ComA. 26

2.4 The basic architecture of MA. 26

2.5 A basic paradigm of assembling ISS-Conductor based components. 28

3.1 A logical view of the functional components in an MA. 32

3.2 A partial activity-transition graph of the example. In the description, actions: Start, DoStep and Stop do not have dependencies on data objects, the action InitSimulation requires a *Setting* object as its input, and the action *Export-Data* has a *Result* object as its output. In the example, the action *DoStep* has a transition with a condition guard: error > Setting.error, which means the action will only be performed when the error is larger than a given bound. . 33

3.3 A simple model of human interaction involved systems. 34

3.4 Capability modelling of the components involved with human interactions. A partial activity-transition graph of Fig. 3.3. The term *InState* describes the state of user activities. 35

3.5 A PT net based model of data production-consumption relation. 37
3.6 A sample scenario for roles Producer_A, Consumer_A and Consumer_B. More examples will be discussed in the next chapter. 39
3.7 A belief-transition graph for deriving the states of neighbour roles. 41
3.8 The action control between an Actor and a conductor, and its reflection in the world model. 42
3.9 The execution states of the actions. 43
3.10 A scenario fragment and its marking graph are shown on the left side. The right side shows a possible execution sequence of role A and B, when they do not apply any concurrency controls. The dashed arrows indicate the marking changes that are perceived by the peer role; the markings in Italic font are invalid. 44
3.11 Data structure for state synchronisation. 46

4.1 Routing spaces and their four default profiles defined for ComAs. Roles are {A, B, C}, the ComAs at Conductor are $\{A_c, B_c, C_c\}$ and the ComA at Actor are $\{A_a, B_a, C_a\}$. The regions for the Conductors use solid lines; and for Actors use dash lines. XMIN, YMIN, XMAX and YMAX are the boundary of the entire region. 53
4.2 A detailed lifecycle of a ComA. 54
4.3 The snapshot of the GUI of a Conductor. 55
4.4 Generating run-time configuration files. 56
4.5 Two components constructed for benchmarking ISS-Conductor. 57
4.6 Action reasoning and update of shared objects in between run-time roles. . . 57
4.7 The delays of the local update and reflection and for the remote update of a shared object. The error bars indicate the standard deviation at each step. . 58
4.8 Remote update of shared objects and the throughput of pure TCP sockets. . 59
4.9 The basic architecture of DASII and the configurations of the experiment. The RTI is executed in fs1, fs2, and nics. 59
4.10 Update delay with different RTI locations. The error bars indicate the standard deviations. 59
4.11 The comparison of the T_R. 60
4.12 The remote update of all eight consumers. 61
4.13 Compare the remote update delay of the 8th consumer and 8 times delay of the first consumer. 61
4.14 The remote update of the first Consumer in the federation in different configurations). 61
4.15 Passing messages to four receivers. The error bars indicate the standard deviations. 62
4.16 A scenario of sending data objects and messages between three component instances. 62
4.17 The influence between object distribution and message passing. The error bars show the standard deviation at each time step. 63
4.18 Remote update delay of different number of attributes. The error bars show the standard deviation at each time step. 63

4.19 Benchmark story. It has three nested scenario nets: T2 (scenario A), T3 (scenario B) and T4 (scenario C). 64
4.20 Scenario A (involved roles: Producer_A, Consumer_A and Consumer_B). Using the publish and subscribe mechanism, a data object can be simultaneously consumed by multiple consumers. 65
4.21 Scenario B (involved roles: Producer_A, Producer_B, Consumer_A). *SbT7* and *SbT8* are two critical transitions. 65
4.22 Scenario C (involved roles: Producer_A, Consumer_A, Producer_C and Consumer_C). 66
4.23 Topologies of the routing spaces of the involved roles. The roles that are not involved in the scenario set their routing spaces using the *away from the others* profile. 66
4.24 The total number of events and the time cost for processing an event. 67
4.25 The time cost by *Producer_A* for each scenario in different execution paradigms. The error bars show the standard deviations. 69
4.26 The total number of state-update messages received by the *Producer_A* in each scenario. 70

5.1 A scenario of simulation based operation planning. 74
5.2 The basic functionality of *Flow_Simulator*. 77
5.3 The basic functionality of *Desktop_VRE*. 78
5.4 A partial activity-transition graph of the C_Flow_Simulator component. See the definition of component capability in section 3.2. 79
5.5 Performance comparison between ISS-Conductor component and legacy implementation. The measurement shows the time cost for one iteration. The error bars indicate the standard deviations. 79
5.6 A layered vision of the functionality of the Desktop_VRE system. 80
5.7 A partial activity-transition of the C_Desktop_VRE component. The term *InState* describes the user activity state. 81
5.8 An activity diagram for Blood_Flow_Studying scenario. The term *UserState* describes the activity state of a user. 82
5.9 A scenario net of the *Blood_Flow_Studying* scenario. 83
5.10 A screen snapshot of the *Blood_Flow_Studying* scenario. The left window shows the interface of the Conductor and the right one is the interface of the *C_Desktop_VRE* component. The flow boundary is a tube with 32×32×64 lattices. The image in the window shows the velocity vectors of the calculated flow field. 84
5.11 Remote update of shared objects when the simulation is executed on multiple processes. The error bars indicate the standard deviations of 100 measurements. 85
5.12 Asynchronous data transmission between the *Surgeon_A* and *Blood_Simulator* component instances. 86
5.13 A comparison of the time interval between two invocations of *Compute* actions. 86
5.14 Basic time costs for updating a *Flow_Data* object. 88
5.15 The idle time between invoking *Refresh_Flow_Data* actions. 89

5.16 The total time for simulation to do 80 iterations computing. 89
5.17 Multiple users explore the *Blood_Flow* object. The term *UserOpinion* describes the opinion choice selected by a user. When both surgeons accept the simulation results, or one of them does not accept, the simulation scenario ends. The high level scenario decides whether to make a new scenario or to finish the entire story. 94
5.18 User's opinions and the chatting interface. 95
5.19 The delay for remotely updating *Blood_Flow* object with one, two and four *Surgeons*. The error bars indicate the standard deviations. The size of data object is 2M bytes. 95
5.20 A scenario template for collaboratively searching optimal bypasses. In the template, *SURGEON_ROLE* and *SIMULATOR_ROLE* are two general role names, which have to be replaced with concrete role names at run time. . . . 97
5.21 Switch shared classes between different instances of a scenario template. . . 98

6.1 A graphical representation of the experiment requirement for a bypass-validation experiment. 107
6.2 Different types of ontologies. 109
6.3 Developing Ontologies using Protégé. 109
6.4 The basic architecture of an agent. 111
6.5 A snapshot of the component management subsystem. 112
6.6 A snapshot of the experiment planning subsystem. 112
6.7 Searching different number of activities from the component collection. . . . 113

References

[1] Thomas J. Schaefer. A transistor-level logic-with-timing simulator for MOS circuits. In *Proceedings of the 22nd ACM/IEEE conference on Design automation*, pages 762–765. ACM Press, 1985.

[2] Milind M. Datar. Enterprise simulation: framework for a strategic application. In *Proceedings of the 32nd conference on Winter simulation*, pages 2010–2014. Society for Computer Simulation International, 2000.

[3] Thomas L. Clarke, editor. *Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment*, volume CR58 of *Critical Reviews of Optical Science and Technology*. SPIE, Bellingham, WA, 1995.

[4] Ernest H. Page and Roger Smith. Introduction to military training simulation: a guide for discrete event simulationists. In *Proceedings of the 30th conference on Winter simulation*, pages 53–60. IEEE Computer Society Press, 1998.

[5] J. F. de Ronde. *Mapping in High Performance Computing - A Case Study in Finite Element Simulation*. PhD thesis, University van Amsterdam, Amsterdam, The Netherlands, (Promoter: Prof. Dr. P. M. A. Sloot), 1998.

[6] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Turan, M. Zingale, and G. Henry. High performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 56. IEEE Computer Society, 2000.

[7] Anu Maria. Introduction to modelling and simulation. In *Proceedings of the 29th conference on Winter simulation*, pages 7–13. ACM Press, 1997.

[8] T. E. Tezduyar, R. Glowinski, J. Liou, T. Nguyen, and S. Poole. Block-iterative finite element computations for incompressible flow problems. In *Proceedings of the 2nd international conference on Supercomputing*, pages 284–294. ACM Press, 1988.

[9] Dirk Bauer and Ronald Peikert. Vortex tracking in scale-space. In *Proceedings of the symposium on Data Visualisation 2002*, pages 233–ff. Eurographics Association, 2002.

[10] Kevin Roe, Duane Stevens, and Carol McCord. High resolution weather modeling for improved fire management. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 48–48. ACM Press, 2001.

[11] Abdel Monim Artoli. *Mesoscopic Computational Haemodynamics*. PhD thesis, Universiteit van Amsterdam, Amsterdam, NL, (Promoter: Prof. Dr. P. M. A. Sloot), 2004.

[12] R. G. Belleman. *Interactive Exploration in Virtual Environments*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, (Promoter: Prof. Dr. P. M. A. Sloot), April 2003.

[13] Elias Houstis, Efstratios Gallopoulos, Randall Bramley, and John Rice. Problem-solving environments for computational science. *IEEE Computational Science & Engineering*, 4(3):18–21, 1997.

[14] Efstratios Gallopoulos, Elias N. Houstis, and John R. Rice. Workshop on problem-solving environments: findings and recommendations. *ACM Computing Surveys (CSUR)*, 27(2):277–279, 1995.

[15] A.S.Z. Belloum, D.L. Groep, Z.W. Hendrikse, L.O. Hertzberger, V. Korkhov, C.T.A.M. de Laat, and D. Vasunin. VLAM-G: A Grid-based virtual laboratory. *Future Generation Computer Systems*, 19(2):209–217, 2003.

[16] C. Johnson, S. Parker, and D. Weinstein. Large-scale computational science applications using the SCIRun problem solving environment. In *Proceedings of Supercomputer*, 2000.

[17] Lemme Group. CtCoq: an environment for mathematical reasoning. *SIGSAM Bull.*, 33(3):21–22, 1999.

[18] Gregor von Laszewski and Ian Foster. Grid infrastructure to support science portals for large scale instruments. In *Proceedings of the Workshop Distributed Computing on the Web (DCW)*. University of Rostock, Germany, June 1999.

[19] H. Afsarmanesh, R.G. Belleman, A.S.Z. Belloum, A. Benabdelkader, J.F.J. van den Brand, G.B. Eijkel, A. Frenkel, C. Garita, D.L. Groep, R.M.A. Heeren, Z.W. Hendrikse, L.O. Hertzberger, J.A. Kaandorp, E.C. Kaletas, V. Korkhov, C.T.A.M. de Laat, P.M.A. Sloot, D. Vasunin, A. Visser, and H.H. Yakali. VLAM-G: A Grid-based Virtual Laboratory. *Scientific Programming: Special Issue on Grid Computing*, 10(2):173–181, 2002.

[20] Upul Obeysekare, Chas Williams, Jim Durbin, Larry Rosenblum, Robert Rosenberg, Fernando Grinstein, Ravi Ramamurthi, Alexandra Landsberg, and William Sandberg. Virtual workbench - a non-immersive virtual environment for visualizing and interacting with 3d objects for scientific visualization. In *Proceedings of the 7th conference on Visualization '96*, pages 345–ff. IEEE Computer Society Press, 1996.

[21] Ersin Cem Karletas. *Scientific information management in collaborative experimentation environments).* PhD thesis, Universiteit van Amsterdam, Amsterdam, NL, (Promoter, Prof. Dr. L. O. Hertzberger), 2004.

[22] Afsarmanesh H., Guevara-Masis V., and Hertzberger L.O. Management of federated information in tele-assistance environments. *Journal on Information Technology in Healthcare*, 2(2):87–108, 2004.

[23] Suad Alagic. The ODMG object model: does it make sense? In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 253–270. ACM Press, 1997.

[24] Jewel Ward. A quantitative analysis of unqualified dublin core metadata element set usage within data providers registered with the open archives initiative. In *Proceedings of the third ACM/IEEE-CS joint conference on Digital libraries*, pages 315–317. IEEE Computer Society, 2003.

[25] Jim Smith, Paul Watson, Sandra de F. Mendes Sampaio, and Norman Paton. Polar: an architecture for a parallel ODMG compliant object database. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 352–359. ACM Press, 2000.

[26] F. Tuijnman and H. Afsarmanesh. Management of shared data in federated co-operative peer environment. *International journal of intelligent and co-operative information systems*, 2(4), 1993.

[27] Ian Foster and Robert L. Grossman. Data integration in a bandwidth-rich world. *Commun. ACM*, 46(11):50–57, 2003.

[28] J. Meng, A. Helal, and Stanley Su. An ad-hoc workflow architecture based on mobile agent and rule-based processing. In *Proceedings of the International Conference on Parallel and Distributed Computing Techniques and Applications*, 2000.

[29] Mor Peleg, Iwei Yeh, and Russ Altman. Modelling Biological Processes using Workflow and Petri Net Models. *Bioinformatics*, 18(6):825–837, 2002.

[30] Candace L. Conwell, Rosemary Enright, and Marcia A. Stutzman. Capability maturity models support of modeling and simulation verification, validation, and accreditation. In *Proceedings of the 32nd conference on Winter simulation*, pages 819–828. Society for Computer Simulation International, 2000.

[31] W. B. Noffsinger, Robert Niedbalski, Michael Blanks, and Niall Emmart. Legacy object modelling speeds software integration. *Commun. ACM*, 41(12):80–89, 1998.

[32] Jeff S. Steinman. Incremental state saving in speedes using C++. In *Proceedings of the 25th conference on Winter simulation*, pages 687–696. ACM Press, 1993.

[33] Philip R. Cohen. Integrated interfaces for decision-support with simulation. In *Proceedings of the 23rd conference on Winter simulation*, pages 1066–1072. IEEE Computer Society, 1991.

[34] Günter Knittel and Wolfgang Straβer. Vizardvisualisation accelerator for real-time display. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 139–146. ACM Press, 1997.

[35] Tim Pattison, Rudi Vernik, Daniel Goodburn, and Matthew Phillips. Rapid assembly and deployment of domain visualisation solutions. In *Australian symposium on Information visualisation*, pages 19–26. Australian Computer Society, Inc., 2001.

[36] G. Cheng, Y. Lu, G. Fox, K. Mills, and T. Haupt. An interactive remote visualisation environment for an electromagnetic scattering simulation on high performance computing system. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 317–326. ACM Press, 1993.

[37] R. G. Belleman and R. Shulakov. High performance distributed simulation for interactive simulated vascular reconstruction. In *P. M. A. Sloot; C. J. K. Tan; J. J. Dongarra and A. G. Hoekstra, editors, Computational Science - ICCS 2002, Proceedings Part III, in series Lecture Notes in Computer Science, vol.2331*, pages 265–274. Springer Verlag, April 2002.

[38] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andr Merzky, Thomas Radke, Edward Seidel, and John Shalf. The Cactus code: A problem solving environment for the Grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 253. IEEE Computer Society, 2000.

[39] K. S. Perumalla and R. M. Fujimoto. Interactive parallel simulations with the Jane framework. *Future Generation Computer Systems*, 17:525, March 2001.

[40] Kyoung S. Park, Yong J. Cho, Naveen K. Krishnaprasad, Chris Scharver, Michael J. Lewis, Jason Leigh, and Andrew E. Johnson. CAVERNsoft G2: a toolkit for high performance tele-immersive collaboration. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 8–15. ACM Press, 2000.

[41] Maarten Boasson. Control systems software. *IEEE Transactions on automatic control*, 38(7):1094–1106, 1993.

[42] Paul Dechering, Rix Groenboom, Edwin de Join, and Jan Tijmen Udding. Formalisation of a software architecture for embedded systems: a process algebra for splice. In *Proceedings of the 32nd Hawaii International Conference on system sciences*, volume 3. IEEE Computer Society, January 1999.

[43] Deborah A. Fullford. Distributed interactive simulation: its past, present, and future. In *Proceedings of the 28th conference on Winter simulation*, pages 179–185. ACM Press, 1996.

[44] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[45] Richard M. Weatherly, Annette L. Wilson, and Sean P. Griffin. Alsp-theory, experience, and future directions. In *Proceedings of the 25th conference on Winter simulation*, pages 1068–1072. ACM Press, 1993.

[46] Larry Mellon and Darrin West. Architectural optimizations to advanced distributed simulation. In *Proceedings of the 27th conference on Winter simulation*, pages 634–641. ACM Press, 1995.

[47] Wentong Cai, Francis B. S. Lee, and L. Chen. 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 Press, 1999.

[48] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*, pages 142–149. ACM Press, 1997.

[49] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 10 April 1998.

[50] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, February, 1997.

[51] Defence Modelling and Simulation Office (DMSO). DMSO RTI commercialisation announcement. In *https://sdc.dmso.mil/announcement.php*, 2003.

[52] H. Zhao and N. D. Georganas. An Approach for Stream Transmission Over HLA-RTI in Distributed Virtual Environments. In *3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pages 67–74, College Park, Maryland, March 1999.

[53] Agostino G. Bruzzone, Roberto Mosca, and Roberto Revetria. Yachts: yet another cooperative high level architecture training software. In *Proceedings of the 33rd conference on Winter simulation*, pages 1619–1623. IEEE Computer Society, 2001.

[54] Ulrich Kelin, Thomas Schulze, and Steffen Straβburger. Traffic simulation based on the high level architecture. In *Proceedings of the 30th conference on Winter simulation*, pages 1095–1104. IEEE Computer Society Press, 1998.

[55] Wentong Cai, Stephen J. Turner, and Boon Ping Gan. Hierarchical federations: an architecture for information hiding. In *Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pages 67–74. IEEE Computer Society, 2001.

[56] Bu-Sung Lee, Wentong Cai, and Junlan Zhou. A causality based time management mechanism for federated simulation. In *Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pages 83–90. IEEE Computer Society Press, 2001.

[57] Rassul Ayani, Farshad Moradi, and Gary Tan. Optimizing cell-size in grid-based DDM. In *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 93–100. IEEE Computer Society, 2000.

[58] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[59] Nicholas Carriero and David Gelernter. Applications experience with linda. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 173–187. ACM Press, 1988.

[60] Silva Filho R. S., Wainer J., E. R. M. Madeira, and C. Ellis. Corba based architecture for large scale workflow. *IEICE Transation on communication*, E83-B No. 5:988–998, 2000.

[61] Francisco J. Gonzá lez Castaño, Javier Vales-Alonso, Miron Livny, Enrique Costa-Montenegro, and Luis Anido-Rifón. Condor Grid computing from mobile handheld devices. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(2):18–27, 2002.

[62] John Durrett and Theron Stimmel. A production-system model of human-computer interaction. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 393–399. ACM Press, 1982.

[63] Ben Shneiderman. *Designing the user interface (2nd ed.): strategies for effective human-computer interaction*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[64] Ben Shneiderman. Putting the human factor into systems development. In *Proceedings of the eighteenth annual computer personnel research conference*, pages 1–13. ACM Press, 1981.

[65] Hurrion R. D. and R. J. Secker. Visual interactive simulation: An aid to decision making. In *Omega 6, 5*, pages 419–426, 1978.

[66] Constance Heitmeyer and Dino Mandrioli. Formal methods for real-time computing: an overview. In *Formal methods for real-time computing*, pages 1–29, West Sussex, England, 1996.

[67] Thom McLean, Richard Fujimoto, and Brad Fitzgibbons. Next Generation Real-Time RTI. In *"Proc 5th IEEE DS-RT" 2001 Fifth IEEE International Workshop on Distributed Simulation and Real Time Applications"*, Cincinnati, Ohio, Aug 2001.

[68] Mostafa A. Bassiouni, Ming-Hsing Chiu, Margaret Loper, Michael Garnsey, and Jim Williams. Performance and reliability analysis of relevance filtering for scalable distributed interactive simulation. *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, 7(3):293–331, 1997.

[69] Robert Macredie, Simon J. E. Taylor, Xiaoning Yu, and Richard Keeble. Virtual reality and simulation: an overview. In *Proceedings of the 28th conference on Winter simulation*, pages 669–674. ACM Press, 1996.

[70] K. A. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. A. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak. The Polder computing environment, a system for interactive distributed simulation. *Concurrency and Computation: Practice and Experience(Special Issue on Grid Computing Environments)*, 14(13–15):1313–1335, 2002.

[71] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[72] Thomas Mastaglio. Developing a large-scale distributed interactive simulation system. In *Proceedings of the 26th conference on Winter simulation*, pages 770–774. Society for Computer Simulation International, 1994.

[73] Christopher D. Carothers, Richard M. Fujimoto, Richard M. Weatherly, and Annette L. Wilson. Design and implementation of HLA time management in the rti version f. 0. In *Proceedings of the 29th conference on Winter simulation*, pages 373–380. ACM Press, 1997.

[74] Richard M. Fujimoto and Richard M. Weatherly. Time management in the DoD high level architecture. In *Proceedings of the tenth workshop on Parallel and distributed simulation*, pages 60–67. IEEE Computer Society Press, 1996.

[75] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[76] C. Szyperski and C. Pfister. Workshop on component oriented programming, summary. In *Special Issues in Object Oriented Programming ECOOP 96 workshop reader*, 1997.

[77] C. Szyperski. Component technology: what, where, and how? In *Proceedings of the 25th international conference on Software engineering*, pages 684–693, 2003.

[78] David C. Luckham, James Vera, and Sigurd Meldal. Key concepts in architecture definition languages. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 2, pages 23–45. Cambridge University Press, 2000.

[79] Joseph A. Heim. Integrating distributed simulation objects. In *Proceedings of the 29th conference on Winter simulation*, pages 532–538. ACM Press, 1997.

[80] S. Calderoni and J. C. Souliè. Jaafaar: A web-based multi-agent toolkit for collective research. *Annals of Software Engineering*, 13(1-4):265–283, 2002.

[81] Gajanana Nadoli and John E. Biegel. Intelligent manufacturing-simulation agents tool (IMSAT). *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, 3(1):42–65, 1993.

[82] David Foulser. Iris explorer: a framework for investigation. *SIGGRAPH Computer Graphics*, 29(2):13–16, 1995.

[83] R. Orfali, D. Harkey, and J Edwards. *The essential distributed objects survival guide*. Wiley, 1996.

[84] John A. Miller, Youngfu Ge, and Junxin Tao. Component-based simulation environments: JSIM as a case study using java beans. In *Proceedings of the 30th conference on Winter simulation*, pages 373–382. IEEE Computer Society Press, 1998.

[85] M. Li, O. F. Rana, M. S. Shields, and D. W. Walker. A wrapper generator for wrapping high performance legacy codes as Java/CORBA components. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 13, 2000.

[86] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)*, 30(1):3–27, 1998.

[87] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 51, August 2000.

[88] S. Parr, A. Radeski, R. Keith-Magee, and J. Wharington. Component-based development extensions to hla. In *Proceedings of Spring Simulation Interoperability Workshop (SISO Spring 2002)*, 2002.

[89] Arjun Cholkar and Philip Koopman. A widely deployable web-based network simulation framework using corba IDL-based APIs. In *Proceedings of the 31st conference on Winter simulation*, pages 1587–1594. ACM Press, 1999.

[90] Michael Pidd, Noelia Oses, and Roger J. Brooks. Component-based simulation on the web? In *Proceedings of the 31st conference on Winter simulation*, pages 1438–1444. ACM Press, 1999.

[91] M. Brasse. A component architecture for federate development description. In *Fall Simulation Interoperability Workshop*, 1999.

[92] George Chin, Jr. , L. Ruby Leung, Karen Schuchardt, and Debbie Gracio. New paradigms in problem solving environments for scientific computing. In *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 39–46. ACM Press, 2002.

[93] John A. Miller, Amit P. Sheth, Krys J. Kochut, Xuzhong Wang, and Arun Murugan. Simulation modelling within workflow technology. In *Proceedings of the 27th conference on Winter simulation*, pages 612–619. ACM Press, 1995.

[94] Philippe Massonet, Yves Deville, and Cèdric Néve. From aose methodology to agent implementation. In *Proceedings of the first international joint conference on Autonomous agents and multi agent systems*, pages 27–34. ACM Press, 2002.

[95] A. Kay. Computer software. *Scientific American*, 251(3):53–59, 1984.

[96] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):1–40, 1996.

[97] M. Wooldridge and N. Jenings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[98] Lisa A. Schaefer. Transport applications: architecture using jini technology for simulation of an agent-based transportation system. In *Proceedings of the 33rd conference on Winter simulation*, pages 1079–1083. IEEE Computer Society, 2001.

[99] Seungman Lee, Amy Pritchett, and David Goldsman. Hybrid agent-based simulation for analyzing the national airspace system. In *Proceedings of the 33rd conference on Winter simulation*, pages 1029–1036. IEEE Computer Society, 2001.

[100] Lamjed Ben Said, Thierry Bouron, and Alexis Drogoul. Agent-based interaction analysis of consumer behaviour. In *Proceedings of the first international joint conference on Autonomous agents and multi agent systems*, pages 184–190. ACM Press, 2002.

[101] P. Terna. Simulation tools for social scientists: Building agent based models with swarm. *Journal of Artificial Societies and Social Simulation*, 1(2), 1998.

[102] Robert Ghanea-Hercock. Assimilation and survival in cyberspace. In *Proceedings of the first international joint conference on Autonomous agents and multi agent systems*, pages 213–214. ACM Press, 2002.

[103] Miles T. Parker. What is Ascape and why should you care? *Journal of Artificial Societies and Social Simulation*, 4(1), 2001.

[104] R. J. Gallimore, N. R. Jennings, H. S. Lamba, C. L. Mason, and B. J. Orenstein. Co-operating agents for 3d scientific data interpretation. *IEEE Trans. on Systems, Man and Cybernetics, Part C*, 29(1):110–126, 2000.

[105] T. J. Norman and N. R. Jennings. Constructing a virtual training laboratory using intelligent agents. *Int. Journal of Continuous Engineering and Life-Long Learning*, 2000.

[106] Y. Yan and S. Ramaswamy. Interactive, agent based, modelling and simulation of virtual manufacturing assemblies. In *Proceedings of the 36th annual conference on Southeast regional conference*, pages 78–87, 1998.

[107] W. Shen, D. Xue, and D. H. Norrie. An agent-based manufacturing enterprise infrastructure for distributed integrated intelligent manufacturing systems. In Hyacinth S. Nwana and Divine T. Ndumu, editors, *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, pages 533–548, London, UK, 1998.

[108] G. Tan, L. Xu, F. Moradi, and S. Taylor. An agent-based ddm for high level architecture. In *Proceedings of the 15th workshop on Parallel and distributed simulation*, pages 75–82, 2001.

[109] L Bölöni. The bond 3 agent system. White paper, http://bond.cs.ucf.edu, 2003.

[110] Steven J. DeRose. XML linking. *ACM Comput. Surv.*, 31(4es):21, 1999.

[111] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM Press, 1994.

[112] L. Bölöni, D. C. Marinescu, J. R. Rice, P. Tsompanopoulou, and E. A. Vavalis. Agent based scientific simulation and modelling. *Concurrency: practice and experience*, 12:845–861, 2000.

[113] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *Ninth IEEE International Symposium on High Performance andDistributed Computing*, 2000.

[114] R. McClatchey and G. Vossen. Workshop on workflow management in scientific and engineering applicationsreport. *SIGMOD Rec.*, 26(4):49–53, 1997.

[115] Jeanine Weissenfels, Michael Gillmann, Olivier Roth, German Shegalov, and Wolfgang Wonner. The mentor-lite prototype: A light-weight workflow management system. In *16th International Conference on Data Engineering*, pages 685–686, February 2000.

[116] Wim Huiskamp, Henk Janssen, and Hans Jense. An hla based flight simulation architecture. In *Proceedings of AIAA Modelling and Simulation Technologies Conference*, August 2000.

[117] Z. Zhao, R. G. Belleman, G. D. van Albada, and P. M. A. Sloot. State update and scenario switch in an agent based solution to constructing interactive simulation systems. In *Proceedings of the Communication Networks and Distributed Systems Model-ing and Simulation Conference*, pages 3–10, San Antonio, US, January 2002.

[118] Z. Zhao, R. G. Belleman, G. D. van Albada, and P. M. A. Sloot. AG-IVE an agent based solution to constructing interactive simulation systems. In *Proceedings of the second International Conference of Computational Science (ICCS02)*, Amsterdam, NL, April 2002.

[119] Message Passing Interface Forum. MPI-2: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 12:1–2, 1998.

[120] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[121] Jon Siegel. OMG overview: CORBA and the OMA in enterprise computing. *Commun. ACM*, 41(10):37–43, 1998.

[122] Shigeru Watanabe. 5-symbol 8-state and 5-symbol 6-state universal turing machines. *J. ACM*, 8(4):476–483, 1961.

[123] James J. Odell. *Advanced Object-Oriented Analysis and Design Using UML*. 1998.

[124] Kari Kuutti and Tuula Arvonen. Identifying potential CSCW applications by means of activity theory concepts: a case example. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 233–240. ACM Press, 1992.

[125] Philip Barnard, Jon May, David Duke, and David Duce. Systems, interactions, and macrotheory. *ACM Transaction Computer-Human Interaction*, 7(2):222–262, 2000.

[126] R. Bastos, D. Dubugras, and A. Ruiz. Extending UML activity diagram for workflow modelling in production systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 291. IEEE Computer Society, 2002.

[127] Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *ICDT*, pages 230–246, 1997.

[128] W. M. P. van der Aalst. The Application of Petri nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[129] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

[130] Ludwik Czaja. Place/transition Petri net evolutions: recording ways, analysis and synthesis. *Fundam. Inf.*, 51(1):43–58, 2002.

[131] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[132] Alois Ferscha. Concurrent execution of timed Petri nets. In *Proceedings of the 26th conference on Winter simulation*, pages 229–236. Society for Computer Simulation International, 1994.

[133] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.

[134] Siegfried I. Mensch and Hans Martin Lipp. Fuzzy specification of finite state machines. In *Proceedings of the conference on European design automation*, pages 622–626. IEEE Computer Society Press, 1990.

[135] Zhigang Wen, Q.H. Mehdi, and N.E. Gough. A new animation approach for visualizing intelligent agent behaviours in a virtual environment. In *Proceedings of sixth International Conference on Information Visualisation (IV'02)*, pages 93–98, July 2002.

[136] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[137] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics Journal.*, online, June 16, 2004.

[138] Kaizar Amin and Gregor von Laszewski. GridAnt: a Grid workflow system. In *Manual at http://www-unix.globus.org/cog/projects/gridant/*, February 2003.

[139] Amzi Inc. Amzi prolog homepage. In *http://www.amzi.com/*, 2002.

[140] (DAS-2). In *The Distributed ASCI Supercomputer 2, Homepage: http://www.cs.vu.nl/das2/*, 2002.

[141] The SurfNet. The surfnet homepage. In *http://www.surfnet.nl/*, 2002.

[142] The MyriCom. The myricom homepage. In *http://www.myri.com/*, 2002.

[143] Kees Verstoep. Discussion on TCP sockets on the DAS II system. In *Internal discussion notes.*, 2004.

[144] Michael W. Vannier, Jeffrey L. Marsh, and James O. Warren. Three dimensional computer graphics for craniofacial surgical planning and evaluation. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 263–273. ACM Press, 1983.

[145] Steven Pieper, Joseph Rosen, and David Zeltzer. Interactive graphics for plastic surgery: a task-level analysis and implementation. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 127–134. ACM Press, 1992.

[146] Kevin Montgomery, Michael Stephanides, Stephen Schendel, and Muriel Ross. A case study using the virtual environment for reconstructive surgery. In *Proceedings of the conference on Visualisation '98*, pages 431–434. IEEE Computer Society Press, 1998.

[147] Carol Bick. Abdominal aortic aneurysm repair. *Nursing Standard*, 15(3):47–52, 2000.

[148] H. G. Burkitt, C. R. G. Quick, and D. Gatt. *Essential Surgery*. Churchill Living Stone, ISBN: 0-443-04805-3, 1996.

[149] Adnan Kastrati, Julinda Mehilli, Stefan Nekolla, Hildegard Bollwein, Stefan Martinoff, Jürgen Pache, Helmut Schühlen, Melchior Seyfarth, Meinrad Gawaz, Franz-JosefNeumann, Josef Dirschinger, Markus Schwaiger, Albert Schömig, , and STOPAMI-3 Study Investigators. A randomized trial comparing myocardial salvage achieved by coronary stenting versus balloon angioplasty in patients with acute myocardial infarction considered ineligible for reperfusion therapy. *Journal of the American College of Cardiology*, 43(5):734–741, 2004.

[150] Daniel Bielser and Markus H. Gross. Interactive simulation of surgical cuts. In *Pacific Graphics 2000 (PG'00)*, page 16, October 2000.

[151] T. P. Grantcharov, V. B. Kristiansen, J. Bendix, L. Bardram, J. Rosenberg, and P. Funch-Jensen. Randomised trial randomised clinical trial of virtual reality simulation for laparoscopic skills training. *British Journal of Surgery*, 91(2):146–150, 2003.

[152] Frederic I. Parke and Mark Friedell. Interactive simulation of biomechanical systems: The kinematics and stress of the human knee. In *Proceedings of the 1978 annual conference*, pages 759–764, 1978.

[153] Y. Zhu, J. X. Chen, S. Xiao, and E. B. MacMahon. 3d knee modelling and biomechanical simulation. *IEEE Computing in Science and Engineering*, 1:82–87, July/August, No. 4, 1999.

[154] Burkhard C. Wünsche, Richard Lobb, and Alistair A. Young. The visualisation of myocardial strain for the improved analysis of cardiac mechanics. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Austalasia and Southe East Asia*, pages 90–99. ACM Press, 2004.

[155] Artoli A. M., Hoekstra A. G., and Sloot P. M. A. Accelerated lattice BGK method for unsteady flow simulations through mach number annealing. *International Journal of Modern Physics C*, 14(6):835–847, 2003.

[156] University of Amsterdam Section Computational Science. Homepage. In *http://www.science.uva.nl/research/scs/*, 2003.

[157] B. D. Kandhai. *Large Scale Lattice-Boltzmann Simulations (Computational Methods and Applications)*. PhD thesis, University van Amsterdam, Amsterdam, The Netherlands, (Promoter: Prof. Dr. P. M. A. Sloot), 1999.

[158] R. G. Belleman and P. M. A. Sloot. The design of dynamic exploration environments for computational steering simulations. In *Proceedings of the SGI Users' Conference 2000, ISBN 83-902363-9-7*, pages 57–74, Academic Computer Centre CYFRONET AGH, Krakow, Poland, 2000.

[159] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In *SIGGRAPH '93 Computer Graphics Conference*, pages 135–142, 1993.

[160] W. Schroeder, K. Martin, B. Lorensen, and Prentice Hall. *The Visualisation Toolkit; An Object-Oriented Approach to 3D Graphics, 2nd edition*. 1997.

[161] The Open PBS homepage. Portable batch system. In *http://www.openpbs.org/*, 2004.

[162] Ulana Legedza and William E. Weihl. Reducing synchronization overhead in parallel simulation. In *Proceedings of the tenth workshop on Parallel and distributed simulation*, pages 86–95. IEEE Computer Society, 1996.

[163] V. Pekar, D. Hempel, G. Kiefer, M. Busch, and J. Weese. Efficient visualisation of large medical image datasets on standard pc hardware. In *Proceedings of the symposium on Data visualisation 2003*, pages 135–140. Eurographics Association, 2003.

[164] Hiromi T. Tanaka, Yasufumi Takama, and Hiroki Wakabayashi. Accuracy-based sampling and reconstruction with adaptive grid for parallel hierarchical tetrahedrization. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 79–86. ACM Press, 2003.

[165] E.V. Zudilova, P.M.A. Sloot, and R.G. Belleman. A multi-modal interface for an interactive simulated vascular reconstruction system. In *Fourth IEEE ACMI'02 International Conference on Multimodal Interfaces*, pages 313–318. IEEE Computer Society, 14-16 October 2002.

[166] H.S.M. Cramer, V. Evers, E.V. Zudilova, and P.M.A. Sloot. Context analysis to inform virtual reality application development. *International Journal of Virtual Reality*, 7(3):177–186, 2004.

[167] Jakob Bardram. Designing for the dynamics of cooperative work activities. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 89–98. ACM Press, 1998.

[168] Richard Furuta and P. David Stotts. Interpreted collaboration protocols and their use in groupware prototyping. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 121–131. ACM Press, 1994.

[169] Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transaction Computer-Human Interaction*, 9(1):1–41, 2002.

[170] Elizabeth A. Hinkelman. Evaluation of collaborative systems using communication actions. *SIGGROUP Bull.*, 20(2):30–33, 1999.

[171] Albert M. Selvin and Maarten Sierhuis. Strategies for collaborative modelling and simulation (workshop session)(abstract only). In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, page 2. ACM Press, 1996.

[172] Tom Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[173] Jörn W. Janneck. Behavioural prediction of time Petri nets with applications to distributed simulation. In *Proceedings High Performance Computing '98*, pages 416–424, 1998.

[174] Sierhuis M and A. M. Selvin. Towards a framework for collaborative modelling and simulation. In *the Workshop on Strategies for Collaborative Modelling and Simulation, CSCW'96 conference*. ACM Press, 1996.

[175] Brad Myers, Jim Hollan, Isabel Cruz, Steve Bryson, Dick Bulterman, Tiziana Catarci, Wayne Citrin, Ephraim Glinert, Jonathan Grudin, and Yannis Ioannidis. Strategic directions in human-computer interaction. *ACM Computing Surveys*, 28(4):794–809, 1996.

[176] Munir Mandviwalla and Lorne Olfman. What do groups need? a proposed set of generic groupware requirements. *ACM Transaction Computer-Human Interaction*, 1(3):245–268, 1994.

[177] Thomas A. Funkhouser. Ring: a client-server system for multi-user virtual environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 85–ff. ACM Press, 1995.

[178] Jutta Willamowski, Francois Chevenet, and Francois Jean-Marie. A development shell for cooperative problem-solving environments. *Math. Comput. Simul.*, 36(4-6):361–379, 1994.

[179] Pedro Garcia, Oriol Montalà, Carles Pairot, Robert Rallo, and Antonio Gömez Skarmeta. MOVE: component groupware foundations for collaborative virtual environments. In *Proceedings of the 4th international conference on Collaborative virtual environments*, pages 55–62. ACM Press, 2002.

[180] Du Li and Richard Muntz. COCA: collaborative objects coordination architecture. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 179–188. ACM Press, 1998.

[181] Yann Laurillau and Laurence Nigay. Clover architecture for groupware. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 236–245. ACM Press, 2002.

[182] Peraphon Sophatsathit and Joseph Urban. Integrating software tool communication within an environment. In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 1070–1075. ACM Press, 1992.

[183] A. Sorgatz and S. Wehmeier. Towards high-performance symbolic computing: using mupad as a problem solving environment. *Mathematics and Computers in Simulation*, 49:235, August 1999.

[184] K. A. Hawick, H. A. James, and P. D. Coddington. A reconfigurable component-based problem solving environment. In *Proceedings of Hawaii International Conference on System Sciences (HICSS-34)*, 2000.

[185] Mikel Lujn. Building an object oriented problem solving environment for the parallel numerical solution of PDEs. In *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 149–150. ACM Press, 2000.

[186] R. M. H. Merks, A. G. Hoekstra, J. A. Kaandorp, and P. M. A. Sloot. Problem solving environment for modelling stony coral morphogenesis. In *P. M. A. Sloot; D. Abrahamson; A. V. Bogdanov; J. J. Dongarra; A. Y. Zomaya and Y. E. Gorbachev, editors, Computational Science - ICCS 2003, Proceedings Part I, in series Lecture Notes in Computer Science, vol.2657*, pages 639–648. Springer-Verlag, Heidelberg, June 2003.

[187] Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

[188] Gary Tan, Yu Hu, and Farshad Moradi. Automatic som compatibility check & fom development. In *7th International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pages 60–67, Delft, the Netherlands, October 2004.

[189] Munindar P. Singh. Distributed enactment of multiagent workflows: temporal logic for web service composition. In *Proceedings of the second international*

joint conference on Autonomous agents and multiagent systems, pages 907–914. ACM Press, 2003.

[190] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in web services. *Commun. ACM*, 46(10):29–34, 2003.

[191] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the eleventh international conference on World Wide Web*, pages 77–88. ACM Press, 2002.

[192] M. Bubak, T. Gubala, M. Kapalka, M. Malawski, and K. Rycerz. Grid service registry for workflow composition framework. In *Proceedings of International Conference on Computational Science, LNCS 3038*, pages 34–41. Springer, June 2004.

[193] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of Grid resource management systems for distributed computing. *International Journal of Software: Practice and Experience (SPE).*, 22(2):135–164, 2002.

[194] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann; 2nd edition, 2003.

[195] The European CrossGrid project. Homepage of the European CrossGrid project. In *http://www.eu-crossgrid.org/*, 2004.

[196] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit devices. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[197] Ken ichi Kurata, Christian Saguez, Gerard Dine, Hiroshi Nakamura, and Vincent Breton. Evaluation of unique sequences on the european data grid. In *Proceedings of the First Asia-Pacific bioinformatics conference on Bioinformatics 2003*, pages 43–52. Australian Computer Society, Inc., 2003.

[198] Nirav H. Kapadia, José; A. B. Fortes, and Mark S. Lundstrom. The purdue university network-computing hubs: running unmodified simulation tools via the www. *ACM Trans. Model. Comput. Simul.*, 10(1):39–57, 2000.

[199] Gloria L. Zũǹiga. Ontology: its transformation from philosophy to information systems. In *Proceedings of the international conference on Formal Ontology in Information Systems*, pages 187–197. ACM Press, 2001.

[200] Sergei Nirenburg and Victor Raskin. Ontological semantics, formal ontology, and ambiguity. In *Proceedings of the international conference on Formal Ontology in Information Systems*, pages 151–161. ACM Press, 2001.

[201] N Guarina. Formal ontology and information systems. In *Proceedings of FOIS'98 Formal ontology in information systems*, June, 1998.

[202] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview. In *W3C Recommendation http://www.w3.org/TR/2004/REC-owl-features-20040210/*, 10 February 2004.

[203] Protégé 2000 project. Protégé home page. In *http://protege.stanford.edu/*, 2004.

[204] The Foundation for Intelligent Physical Agents. Homepage of FIPA. In *http://www.fipa.org/*, 2004.

[205] Volker Haarslev and Ralf Möller. Description of the RACER system and its applications. In *Proceedings International Workshop on Description Logics (DL-2001)*, August 2001.

[206] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, December 2003.

[207] The Jgraph Ltd. Professional open source of Java Graph Visualisation. In *Homepage of Jgraph: http://www.jgraph.com/*, 2004.

[208] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 13, August 1999.

[209] T. Drashansky, A. Joshi, and J. Rice. SciAgents – an agent based environment for distributed, co-operative scientific computing. In *Proceedings of the 7th International Conference of Tools with Artificial Intelligence (Los Alamitos, CA), IEEE Computer Soc.*, pages 452–459., 1995.

[210] Christopher Stone. Software standardization: how the Object Management Group changed the model. *StandardView*, 3(3):85–89, 1995.

[211] K. Z. Zajac, A. Tirado-Ramos, Z. Zhao, P. M. A. Sloot, and M. Bubak. Grid services for HLA-based distributed simulation frameworks. In *F. Fernández Rivera; M. Bubak; A. Gómez Tato and R. Doallo, editors, First European Across Grids Conference*, pages 147–154. Springer-Verlag, Heidelberg, February 2003.

[212] K.Z. Zajac, M. Bubak, M. Malawski, and P.M.A. Sloot. Towards a grid management system for HLA-based interactive simulations. In *S.J. Turner and S.J.E. Taylor, editors, Proceedings Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2003)*, pages 4–11. IEEE Computer Society, October 2003.

[213] Katarzyna Rycerz, Marian Bubak, Maciej Malawski, and Peter Sloot. A framework for hla–based interactive simulations on the Grid. *Simulation transaction, Special Issue: Applications of Parallel and Distributed Simulation in Industry.*, submitted.

[214] L. O. Hertzberger. Introduction of VLAM-G and VL-E. In *Internal seminar*, 2004.

Nederlandse Samenvatting

Interactieve simulaties spelen in toenemende mate een belangrijke rol in het wetenschappelijk onderzoek: in vergelijking met conventionele simulaties maken zij het mogelijk parameterruimten efficiënter te onderzoeken en de berekeningen nauwkeuriger aan te sturen. Op deze manier helpen ze ook om het gebruik van hulpbronnen, zowel voor verwerking als voor opslag te optimaliseren en de communicatie efficiënter te maken. Binnen een interactief simulatiesysteem (ISS) kunnen we meestal de volgende componenten onderscheiden: een of meer simulatie-modules die de eigenlijke berekeningen uitvoeren, een of meer visualisatie-modules die de resultaten van de simulatie meteen voor de gebruiker zichtbaar maken, en een of meer interactie-modules waarmee de gebruiker het simulatie- en visualisatie-proces kan sturen. De visualisatie- en interactie-modules worden veelal gecombineerd. De ontwikkeling en integratie van simulatie- en visualisatiekernels is echter kostbaar. Traditionele ontwikkelingsmethoden resulteren in het algemeen in systemen met een sterke koppeling tussen de systeemfunctionaliteit en de applicatieafhankelijke logische besturing en zijn hierdoor moeilijk aan te passen voor andere problemen. Deze hoge kosten en geringe flexibiliteit hinderen de invoering van dergelijke systemen. Bestaande softwarearchitecturen en middleware richten zich met name op de laag-niveau koppeling van systeemcomponenten, en niet expliciet op “rapid prototyping” van ISS of op een flexibele aansturing van het systeemgedrag.

In dit proefschrift stellen we dat de scheiding van de applicatieafhankelijke logische aansturing en de systeemfunctionaliteit een cruciale stap is naar een verbeterd ontwikkelingsproces voor ISS. We hebben “Interactive Simulation System Conductor” ontwikkeld, een architectuur voor de samenbouw van componenten met behulp van 'agents' (agenten), die een dergelijke scheiding realiseert. Deze architectuur vangt de intelligentie voor de besturing van het geïntegreerde systeem in verschillende rollen voor agenten en maakt het mogelijke met deze agenten een gelaagde verbindingsstructuur op te zetten tussen de componenten. Er is een “proof of concept” implementatie gebouwd op basis van de “High Level Architecture” (HLA), een specifiek voor gedistribueerde simulatie-systemen ontworpen middleware.

De basisarchitectuur van ISS-Conductor wordt in het tweede hoofdstuk gepresenteerd. Door de reken-kernels en ondersteunende structuren van de hoofdmodules van een ISS, zoals simulatie, visualisatie en interactie in te kapselen, staat het wetenschappers toe op een hoog niveau simulatie-experimenten op te zetten zonder zich bezig te hoeven houden met de laag-niveau systeemintegratie. Binnen de architectuur leggen “Communicatieagenten” (ComA) de basis voor de interoperabiliteit van

componenten; “Moduleagenten” (MA) orkestreren het gedrag van het systeem tijdens de uitvoering. De “Run-Time Infrastructure” (RTI) van HLA wordt gebruikt als de “software bus” waarmee alle componenten worden samengebouwd.
In het derde hoofdstuk bespreken we het functionele ontwerp van ISS-Conductor. In een ISS-Conductor component stuurt de MA het gedrag van die component, gebruik makend van een redeneer-systeem; dit systeem heeft in zijn “knowledge base” kennis van zowel de functionaliteit van de diverse componenten, als van de voor de applicatie specifieke constraints op de interacties. De functionaliteit van een component (“capability”) wordt gemodelleerd met een eindige automaat die samen met die van de andere componenten kan worden geprogrammeerd, gebruik makend van een mechanisme gebaseerd op een Petri net (“scenario net”). Tijdens de uitvoering interpreteren de MA’s van de verschillende componenten gezamenlijk de constraints op de interactie en besturen op deze manier het gedrag van het systeem als geheel.
Hoofdstuk vier is gewijd aan de details van de implementatie en de performance karakteristieken van ISS-Conductor. De ComA koppelt naar de diensten van de HLA RTI die verantwoordelijk zijn voor het delen van gegevens en het versturen van boodschappen. Het redeneer-systeem van de MA is in Prolog geschreven. Op basis van onze metingen aan de prestaties van onze implementatie kunnen we concluderen dat de Communicatieagenten een acceptabele overhead toevoegen aan de RTI. Applicaties gebaseerd op ISS-Conductor kunnen voor grote dataobjecten een netwerkutilisatie behalen, vergelijkbaar aan die van zuivere TCP sockets. De logische besturing van het redeneersysteem veroorzaakt ook slechts een kleine overhead.
In het vijfde hoofdstuk gebruiken we de ISS-Conductor architectuur om een prototype te maken voor een interactieve simulatieomgeving voor het plannen van vasculaire operaties. We bespreken de procedures voor de ontwikkeling van een ISS-Conductor component in detail, evenals die waarmee de componenten worden samengebouwd tot een interactief systeem. Voor een aantal testcases bespreken we de aansturing op het scenario-niveau, automatische tuning van de prestaties in een lopend systeem en de ondersteuning voor interactieve samenwerking. De experimentele resultaten tonen aan dat ISS-Conductor slechts een kleine overhead toevoegt aan reeds bestaande reken-kernels.
In het zesde hoofdstuk bespreken we de mogelijkheid componenten die aan het ISS-Conductor model voldoen te gebruiken als software binnen een zogenaamd “Problem Solving Environment”. Een van de cruciale problemen die we daarbij onderzoeken is hoe de componenten voor een interactief simulatie-experiment automatisch gevonden en samengevoegd kunnen worden. In dit hoofdstuk wordt ISS-Studio beschreven, een op Java gebaseerd prototype voor een dergelijke omgeving.
In het zevende en laatste hoofdstuk wordt een samenvatting gegeven van het proefschrift en worden de resultaten geplaatst in het licht van de verwachte toekomstige ontwikkelingen op het gebied van interactieve simulatiesystemen.

Publications

[1] P. Kommers and Z. Zhao. Conceptual support with virtual reality in web-based learning. *International Journal of Continuing Engineering Education and Life-Long Learning*, 8(1), 1998.

[2] R. G. Belleman, Z. Zhao, G. D. van Albada, and P. M. A. Sloot. Design considerations for the construction of immersive dynamic exploration environments. In *L. J. van Vliet; J. W. J. Heijnsdijk; T. Kielmann and P. M. W. Knijnenburg, editors, ASCI 2000, Proceedings of the sixth annual conference of the Advanced School for Computing and Imaging*, pages 195–201, the Netherlands, June 2000.

[3] Z. Zhao, R. G. Belleman, G. D. van Albada, and P. M. A. Sloot. System integration for interactive simulation systems using intelligent agents. In *R. L. Lagendijk; J. W. J. Heijnsdijk; A. D. Pimentel and M. H. F. Wilkinson, editors, Proceedings of the 7th annual conference of the Advanced School for Computing and Imaging*, pages 399–406, the Netherlands, May 2001.

[4] Z. Zhao, R. G. Belleman, G. D. van Albada, and P. M. A. Sloot. State update and scenario switch in an agent based solution to constructing interactive simulation systems. In *Proceedings of the Communication Networks and Distributed Systems Model-ing and Simulation Conference*, pages 3–10, San Antonio, US, January 2002.

[5] Z. Zhao, R. G. Belleman, G. D. van Albada, and P. M. A. Sloot. AG-IVE an agent based solution to constructing interactive simulation systems. In *Proceedings of the second International Conference of Computational Science (ICCS02)*, Amsterdam, NL, April 2002.

[6] Z. Zhao, R. G. Belleman, G. D. van Albada, and P. M. A. Sloot. Reusability and efficiency in constructing interactive simulation systems. In *E.F. Deprettere; A.S.Z. Belloum; J.W.J. Heijnsdijk and F. van der Stappen, editors, ASCI 2002, Proceedings of the eighth annual conference of the Advanced School for Computing and Imaging, Delft*, pages 268–275, June 2002.

[7] Z. Zhao, G. D. van Albada and P. M. A. Sloot. Interaction scenario: Orchestrating agents in a multi-agent system. In *J. -P. Muller and M. -M. Seidel, editors, Proceedings of the 4th workshop on Agent-Based Simulation, ISBN 3-936-150-25-7*, pages 155–160, Montpellier, France, April 2003.

[8] A. Tirado-Ramos, K.Z. Zajac, Z. Zhao, P.M.A. Sloot, G.D. van Albada, and M. Bubak. Experimental Grid access for dynamic discovery and data transfer in distributed interactive simulation systems. In *P.M.A. Sloot; D. Abrahamson; A.V. Bogdanov; J.J. Dongarra; A.Y. Zomaya and Y.E. Gorbachev, editors, Computational Science - ICCS 2003, Melbourne, Australia and St. Petersburg, Russia, Proceedings Part I, in series Lecture Notes in Computer Science*, pages 284–292. Springer Verlag, June, 2003.

[9] Z. Zhao, A. Tirado-Ramos, K.Z. Zajac, G.D. van Albada, P.M.A. Sloot. ISS-Studio: a prototype for a user-friendly tool for designing interactive experiments in Problem Solving Environments. In *P.M.A. Sloot; D. Abrahamson; A.V. Bogdanov; J.J. Dongarra; A.Y. Zomaya and Y.E. Gorbachev, editors, Computational Science - ICCS 2003, Melbourne, Australia and St. Petersburg, Russia, Proceedings Part I, in series Lecture Notes in Computer Science, vol. 2657*, pages. 679-688. Springer Verlag, June 2003.

[10] K.Z. Zajac, A. Tirado-Ramos, Z. Zhao, P.M.A. Sloot, and M. Bubak. Grid services for HLA-based distributed simulation frameworks. In *F. Fernández Rivera; M. Bubak; A. Gómez Tato and R. Doallo, editors, First European Across Grids Conference, Santiago de Compostela, Spain.*, pages 147–154. Springer Verlag, February 2003.

[11] Z. Zhao, G. D. van Albada, P.M.A. Sloot. A Layered framework for constructing interactive simulation systems. *ICECCS 2005, Int'l Conf. on Eng. of Complex Compute Systems.* (Submitted.)

[12] Z. Zhao, G. D. van Albada, P.M.A. Sloot. ISS-Conductor: an agent based architecture for interactive simulation systems. *Concurrency: Practice and Experience.* (Submitted.)

[13] Z. Zhao, G. D. van Albada, P.M.A. Sloot. Agent based flow control for HLA components. *Simulation transaction, Special Issue: Agent directed simulation.* (Accepted.)

Index

Common Component Architecture(CCA), 16
- SIDL, 16, 102

Common Object Request Broker Architecture(CORBA), 10, 57
- CCM, 28
- TAO, 10

Grid
- Computational Grid, 3
- OGSA, 6
- VO, 3

High Level Architecture(HLA), 45, 53, 75, 102
- ALSP, 10
- DARPA, 10
- DIS, 10
- DoD, 9
- Federate, 10
- FOM, 10, 102
- LBTS, 14
- libRTI, 10
- OMT, 76
- RTI, 10, 53, 75
- SIMNET, 10
- SOM, 10, 102
- SSA, 16, 29, 91

Interactive Simulation System(ISS), 1
- Computer Simulation, 1
  - PDES, 3
- HPC, 3
  - MPI, 16, 77
  - PVM, 16
- PSE, 1, 5, 101
  - ODMG, 5
  - VLAM-G, 5, 103
- Scientific Visualisation, 4
  - DEE, 4
  - Virtual Reality, 4

ISS-Conductor, 20, 23, 31, 51, 74, 101
- Actor, 25
- Capability, 27, 32
- ComA, 25, 42, 51, 77
- Conductor, 25
- Critical Transition, 44
- FSM, 32
- ISS-Studio, 21, 101
- MA, 25, 31, 52, 88
- Petri net, 35, 91, 101
- Role, 28
- Scenario net, 37, 83
- Story, 27, 31, 53, 77, 103

Acknowledgments

I would like to take this opportunity to thank everybody I have worked with and met during the period of my Ph.D. journey. Without the help from them, this thesis would not come true.

First, I want to thank my promoter, professor Peter Sloot, for bringing me into the exciting field of interactive simulation and high performance computing, and for offering me opportunity to work in his group. Not only the knowledge in modelling and simulation, but also the open mind and critical attitude in science, Peter taught me many things, which I think will be invaluable wealth for my future life. Thanks Peter for all your support.

Without the enormous amount of help from my co-promoter, dr. Dick van Albada, the thesis would not look like this and at least I would have spent much longer time in wandering around the subject. Thanks Dick, for the patience you always had for me during the discussions, and also for the time you spent checking my thesis and preparing a Dutch summary.

Of course I will not forget how my life started in Amsterdam, that was in a cold raining evening, the airplane was delayed, and my luggage was lost. It was Alfons who picked me up in that middle night from the airport and settled me down. Thanks Alfons, also for the help you gave me during my first year in the SCS group.

I want to acknowledge many people for their help when I was working on the subject of Virtual Reality and Scientific Visualisation, although I have latterly shifted a lot from there. Piet Kommers, my former supervisor in University of Twente, invited me as a visiting scholar to work with him for the project of Virtual Reality and distance education. This opportunity turns a new page of my life in the Netherlands. Thanks Piet for all your help. In SCS group, I learned a lot from dr. Jaap Kaandorp about Scientific Visualisation and enjoyed in assisting him for giving courses. I am grateful to Robert Belleman for all the help of VR related techniques and for valuable suggestions on my research. More importantly, without him, I would not get chance to access some nice software (you knows what I mean, Rob). Thanks Elena for nice discussions on human computer interaction. Thanks Dmitry, Vladimir and other Russian colleagues helped me in some experiments in CAVE for studying human behaviour. In particular, I would like to thank Dennis for customising his desktop VRE

program for my thesis experiments and Roman for measuring the performance of his NT software and for allowing me to use it in my paper.

I am grateful to all co-authors of my publication and their pleasant co-operation, in particular Alfredo and Katarzyna.

I would like to thank other members (past and present) of the SCS group as well. As the most stable user of room 219, I have sit in that office for more than 5 years. I really enjoyed the atmosphere of multi culture from all present and former officemates: Artoli is from Sudan, Judhi is from Indonesia, David is from Holland, Tomasz is from Poland, Jordi is from Spain, Yves is from Cameroon and Maxim is from Russia. SCS is a group whose members often work on substantially different subjects, for instance, Michael, Alessia, Evghenii and Mark work on Astronomy, Michael works on medical applications, Kamil works on discrete event simulation, Lilit works on flow simulation and Jiangjun works on biological simulation; however, we can always be perfectly glued by the cosy air of lunch chatting. Although Roeland, Piero, Martin, Benno, Arjen and Edwin left this group, and Lera does not physically work here so often, I will certainly not forget them. Specially, thanks Zeger for helping me to move. Thanks other colleagues Breanndán, Walter, Simon, Drona, Berry and Bas for help. I wish you all the best, and good luck with your work and research.

I also received help from friends from the other groups. I am grateful to Huang Zhisheng, an expert in artificial intelligence, for giving me critical and very helpful comments on my Prolog programming and on my understanding on agent technologies. And I appreciate all types of help received from Alban, Ersin and Frank.

I want to express my sincerely appreciation to the members of my reading committee for their invaluable comments and help on improving the quality of the manuscript: Maarten Boasson, Frances Brazier, Chengzheng Sun, Zhiwei Xu, Hamideh Afsarmanesh and Marian Bubak.

Of course, I will not forget the help from the people at the secretariat, system support group, personnel and financial department and the library: Erik, Coco, Marianne, Virginie, Saskia, Jacqueline, Frans, Heleen and many others.

The Chinese community constitutes an important part of my social life. I am grateful to Mrs Xue Hanqing, I always learn lots of educational things from her about how to live and work abroad. I get appreciated support from Mr. Tong Guangwu, Mr. Dong Huiqing and Mr. Wu Liansheng for organising the social events for Chinese students. Dai Jiapei, Mang Rui, Yang Xiaoxian, Ni Yongfeng, Wang Chao, Wu Wenhua, Hu Xiaofeng, He Qiong, Huo Ran, Zhou Yinwei, Li Ting, Wang Jing, Pang Jun, Qiu Huaning, Qing Fang, Qiu Guangzhong, Wang Jinhua, Cui Jiajia and many others friends gave me great support when I run the Chinese student association. Thanks all of you!

I am grateful to Renate and Petra for their help during my stay in their house, thanks for giving me the feeling of a family. Thanks Zilei for designing the cover of the thesis. And thanks Emmy, Angélique and Pauline for the help when receiving Chinese CCTV crew, and Fris and Rena for helping the photo competition of the student community.

When in hard time of my life, my parents Zhao Jun and He Fengbao, and my parents in law Xu Fukang and Zhang Hongping always support and encourage me and let me feel I am the best. My wife, Yan, quitted her job and joined my life in Holland; without her help, I will not possibly manage all the things of the thesis. I appreciate all the support from Gu Jingliang, Zhao Chunlan, Xuan Xiaofeng and Zhao Zhihua. I really miss Xuan Yi and Gu Xian.

Thanks many other friends who have not been listed.

Zhiming Zhao
Amsterdam, October 23, 2004