An agent based architecture for constructing Interactive Simulation Systems
Zhao, Z.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

Download date: 15 Mar 2019
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 import-
Introduction

A 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. predicting 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.](image)

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.

![Data Flow Diagram](image)

**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 (PSEs) 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. Kaleta categorises 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.
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.](image)

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

*We will have more discussion on this term in 1.6.2.*
Interactive Simulation (DIS)\(^1\) 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\(^2\).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{federation-diagram.png}
\caption{Distributed federates and the Runtime infrastructure (RTI).}
\end{figure}

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 stand-

---

\(^1\)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.

\(^2\)After September 2002, commercial companies are also allowed to implement and realise the RTI software [51].
ard 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 are modelled as an abstract description of the processes in a scientific experiment, 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 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. Hurron [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
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 of distributed modules must be co-ordinated: each simulation should treat its time

\[\text{The DoD's RTI does not support hard real-time distributed simulations [67].}\]
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 oriented, and agent oriented engineering, brought contributions to designing ISSs.
1.6 Engineering methodologies

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 Inter-
face 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 SIMULTAA NN 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 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, 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
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 human interactions at the system behaviour level. This leads to the statement of our research problem.

Figure 1.7: Summary of available architectures.

<table>
<thead>
<tr>
<th>Wrapping</th>
<th>Interoperability</th>
<th>Real time interaction</th>
<th>Interface description</th>
<th>Activity orchestration</th>
<th>Composition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLA</td>
<td>Local RTI lib.</td>
<td>RTI.</td>
<td>Soft/with</td>
<td>Simulation Object</td>
<td>Federation</td>
<td>Suitable for interoperability between distributed simulation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>time management</td>
<td>Model.</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Model.</td>
<td></td>
</tr>
<tr>
<td>SSA</td>
<td>Based on HLA.</td>
<td>RTI, and RCI.</td>
<td>Soft./HLA based.</td>
<td>Simulation Object</td>
<td>Simulation</td>
<td>Based on HLA, with multi-level communication, and a centralised scenario manager.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Model.</td>
<td>scenario.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In the federate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>functionality.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In the functionality</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>of application processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inside Bond agents.</td>
<td>(Finite state</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>machine based).</td>
<td></td>
</tr>
</tbody>
</table>

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.
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.