An agent based architecture for constructing Interactive Simulation Systems

Zhao, Z.

Citation for published version (APA):
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.

![Diagram](image.png)

**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 Introduction

5.1.2 Goal of the chapter

Two packages: a flow simulator (*FlowSimulator*) 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 definition of data classes, activities and their dependencies and 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.

![flow simulator diagram](image)

**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-
realism 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.

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
5.2 From Legacy systems to reusable components

<table>
<thead>
<tr>
<th>Internal class</th>
<th>Shared class Flow_Setting</th>
<th>Shared class Flow_Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>State_Monitor</td>
<td>long iteration;</td>
<td>double ***Velocity,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double ***Pressure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>Get_Simulation_Setting</th>
<th>Export_Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(succeed)</td>
<td>(succeed)</td>
</tr>
<tr>
<td></td>
<td>(iteration=MaxIter)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(succeed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\text{Figure 5.4: A partial activity-transition graph of the C.Flow.Simulator component. See the definition of component capability in section 3.2.}

\text{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.

\text{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 sub-tasks 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. Refresh 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.
5.2 From Legacy systems to reusable components

![Diagram of activity-transition of the C.Desktop.VRE component.](image)

**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 runtime 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 the 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.

![Activity Diagram](image)

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 runtime 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].

![Diagram](attachment:image.png)

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
5.3 Coupling component instances

32×32×64 (lattice units)\(^\dagger\).

![Image](image)

**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

\(^\dagger\)The size of the objects when using these three geometrical structures are 512KB, 1MB and 2MB respectively.
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\(^5\). 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\(^6\). \(T_c\) is related to the number of the lattice points in each simulation process and the number of the iterations in each cycle; \(T_e\), \(T_r\), and \(T_f\) 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

\(^{5}\)The updates of a shared object is buffered in the ComA of the receiver role.

\(^{6}\)We can see this from the experiments in the previous chapter.
quality of 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 $T_c$ and $T_e$. 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.
5.5 Collaborative interaction in an ISS

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 5.15: The idle time between invoking RefreshFlowData actions.](image1)

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

![Figure 5.16: The total time for simulation to do 80 iterations computing.](image2)

**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. WYSIWIS 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

\[\text{What you see is what I see.}\]
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\textsuperscript{\textdaggerdbl}\textdaggerdouble 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.

\textsuperscript{\textdaggerdbl}\textdaggerdouble 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.
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. An easy way for the roles in different scenario instances to maintain the internal data objects is to declare them as data classes. 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
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 WYSIWIS 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.
Rapid Prototyping of a surgical pre-operative planning environment