Experiences with component-oriented technologies in nuclear power plant simulators

Manuel Díaz, Daniel Garrido, Sergio Romero*,†, Bartolomé Rubio, Enrique Soler and José M. Troya

Department of Languages and Computer Science, University of Málaga, 29071 Spain

SUMMARY

This paper proposes the application of modern component-oriented technologies to the development of nuclear power plant simulators. On the one hand, as a significant improvement on previous simulators, the new kernel is based on the Common Component Architecture (CCA). The use of such a high-performance computing oriented component technology, together with a novel algorithm to automatically resolve simulation data dependencies, allows the efficient execution of both parallel and sequential simulation models. On the other hand, RT-CORBA is employed in the development of the rest of the applications that comprise the simulator. This real-time communication middleware not only makes the management of communications easier, but also provides the applications with real-time capabilities. Software components used in these two ways, simulation models integrating the kernel and distributed applications from which the simulator is comprised, improve the evolution and maintenance of the entire system, as well as promoting code reusability in other projects. Copyright © 2006 John Wiley & Sons, Ltd.

Received 16 June 2005; Revised 10 November 2005; Accepted 5 January 2006

KEY WORDS: component-oriented programming; scientific computing; nuclear power plant simulator; CCA; CORBA

1. INTRODUCTION

Simulators are especially important in the context of nuclear power plants since they can predict the plant status when facing different situations that can occur in their daily operations. In this sense, a fast response is required and performance becomes a major factor. Besides that, simulators can be used as

*Correspondence to: Sergio Romero, Department of Languages and Computer Science, University of Málaga, 29071 Spain.
†E-mail: sromero@lcc.uma.es

Copyright © 2006 John Wiley & Sons, Ltd.
training tools for future operators, allowing the practice of both normal and emergency situations in a safe way. To summarize, a pressurized water reactor (PWR) plant consists of a vessel, containing the nuclear reactor, steam generators and hydraulic loops made up of pipes and pumps through which water and steam flow. The reactor produces heat that is carried by pressurized water to the steam generators. They vaporize the water in a secondary loop to drive the turbine, which produces electricity.

In order to simulate this operation in a computer system, a collection of tools and applications used for many different functions comprise the simulator software architecture. In this context, the most important application is the simulator kernel, which is responsible for executing scientific codes implementing detailed mathematical models of the power plant physical subsystems. There is a wide range of these simulation models, from computationally intensive complex models such as TRAC [1] (thermo hydraulic model) or NEMO (neutronic model) to simpler models simulating, for example, the operation of a valve.

In previous work, we have collaborated with the company Tecnatom S.A. [2] in the development and maintenance of different simulators currently used in several power plants located in Spain, Germany and Mexico [3,4]. The kernel’s of these simulators were programmed in a classical style, as all simulation models were coded as FORTRAN and C procedures, statically linked together into the kernel. This approach, however, leads to serious limitations from the software engineering viewpoint. For example, it is very usual for a simulation model to read or update data variables computed by other models. In order to resolve these types of data dependencies, all shared data were declared as global variables allowing access from any procedure. Due to the programming techniques used, common actions such as modification, substitution or integration of new models into the simulator usually turned into tedious tasks. Furthermore, both source code version management and reusability of scientific code in other simulators were also very limited.

Component-Based Software Engineering (CBSE) is a modern methodology that proposes the construction of applications by plugging in standalone software components [5]. Based on component interoperability, this programming style allows the creation of more flexible and adaptable software. Nuclear power plant simulators can take advantage of componentization in order to improve evolution, maintenance and the software life cycle. However, parallelism and high performance, which are requirements of fundamental importance for a simulator kernel that needs to execute hundreds of simulation models in acceptable time, are not taken into account by component standards and implementations such as OMG Corba Component Model (CCM) [6], Microsoft’s DCOM [7], Sun Java Beans and Enterprise Java Beans [8,9]. They also have trouble encapsulating an existing scientific application (which might itself be a parallel or distributed application) into a component. Recently, some efforts are being made in order to incorporate component technologies into the high-performance computing area, which is traditionally based on classical programming techniques and languages such as FORTRAN and, more recently, Java and C++ [10]. In this sense, ASSIST [11] is focused on high-level programmability and software productivity for complex multi-disciplinary applications, including data-intensive and interactive software. SBASCO [12] is oriented to the efficient development of parallel and distributed numerical applications. A large effort is currently being devoted by the Common Component Architecture (CCA) forum [13] to define a standard component architecture for high-performance computing.

In this paper, we present a complete software environment focused on the simulation of nuclear power plant control rooms. Specifically, we expose our experiences obtained from the development of complex simulators combining two new component-based technologies: the architecture proposed by CCA and the OMG Real-time CORBA extension (RT-CORBA) [14].

We focus on the idea that a component-based simulator kernel can solve many software maintenance related problems that appeared in previous versions of these applications. In our proposal, simulation models are encapsulated into software components making it possible to construct different kernels by selecting these components from a simulation model repository. They are connected to a central manager component that is in charge of controlling the kernel execution.

Apart from componentization, another additional aspect has to be taken into account to improve the system. Previous simulation models were implemented through sequential procedures. However, scientific codes from some of the most computationally intensive models can be parallelized to be run on multiple processors with the aim of reducing their execution time. For example, in [15] a parallelized version of the thermo-hydraulic code encapsulated into the TRAC simulation model is described. Parallel execution of this model is especially important since it represents about 80% of the total simulation time. Aiming to combine both componentization and parallelization into the new simulator kernel, we have based its development on CCA. The way our simulator kernel can be componentized and parallelized following a Single Program Multiple Data (SPMD) programming style, fits perfectly the Single Component Multiple Data (SCMD) parallel execution model provided by the CCA-compliant Ccaffeine framework [16]. In this scenario, simulation data will be distributed over the different processes and so, parallel communications may be needed to supply simulation models with their requested variables. Based on a straightforward communication protocol, we use an efficient algorithm that automatically resolves all types of data dependencies, releasing the programmer from this task and, as a consequence, making the development process easier.

The entire simulator constitutes a distributed software system in which semi-independent applications, executed in different network nodes, communicate with the kernel for many different purposes, such as debugging the simulation process, changing simulation aspects like cycling time or recording (and recovering) the simulation state including all significant variables in real-time. The Common Object Request Broker Architecture (CORBA) [17] is the core of the Object Management Architecture described by the Object Management Group (OMG) [18] for the building of distributed applications. The use of CORBA in the simulator development facilitates the management of communications as well as allowing independence of platforms, operating systems and programming languages. Applications in the simulator context can be treated as distributed components in an environment that promotes code reusability and improves software maintenance. Nevertheless, standard CORBA Object Request Brokers (ORBs) traditionally only support best-effort capacities without temporal guarantees. Since these simulators must provide predictable temporal behavior, we have used the real-time CORBA specification (RT-CORBA). Other interesting approaches for developing distributed real-time simulators are based on the TMO model [19] and HLA [20].

The rest of the paper is structured as follows. An overview of the simulator architecture including the most outstanding tools and applications is presented in Section 2. Section 3 provides some of the basic features of used technologies. In Section 4, both the CCA-compliant simulator kernel and the model on which it is based are described. Implementation decisions about employed communication mechanisms are discussed in Section 5. The paper finishes with some conclusions.

2. SIMULATOR OVERVIEW

The simulation projects of Tecnatom S.A. usually include two simulators that influence on hardware and software architectures.
— Full Scope Simulator (FSS) is an exact replica of the power plant control room, taking care of each and every detail, from physical artifacts such as furniture, control panels, etc. to software simulating the applications running in the room (Figure 1, left).
— Interactive Graphic Simulator (IGS) allows operator training through graphic applications (Figure 1, right).

The main high-level hardware elements of FSS and IGS are as follows:

— Simulation Computers are responsible for the simulation process executing the simulation models and providing data to the rest of the software and hardware components.
— Depending on the power plant, there will be additional Hardware Subsystems included in the simulator.
— The Instructor Console (CDI; from the Spanish ‘Consola del Instructor’) is used by the instructor of the simulation sessions and it allows the creation of different scenarios that have to be solved by the students.
— Physical Panels are exact replicas of those existing in the control room. Operators of the power plant carry out their actions mainly through these panels, with hundreds of indicators, hardware keyboards, etc.
— IGS simulators additionally include the hardware needed by Student Workstations that basically allow the practice of any simulation area in a comfortable way with graphical applications and several monitors for each student.

The simulator software architecture comprises a collection of distributed applications interacting with each other through the high-level communications mechanisms of CORBA. Because of the system size and the heterogeneity of the involved applications, the development environment includes different platforms such as Unix, Linux and Windows, with different programming languages such as C++, Java and FORTRAN. As a main goal, all applications and libraries have been developed in
such a way that they could be reused in other simulators. Due to the implementation of a suitable communication infrastructure, new applications can be added to the simulator without modifications in the software architecture.

The system can be mainly divided into two differentiated parts, as shown in Figure 2. The first part is comprised of the simulator kernel together with SimCorba (a CORBA communication layer). This part acts as a simulation server offering a set of services to the rest of the client tools and on-line applications, which constitute the second part of the simulator.

The parallel Simulator Kernel is in charge of executing the simulation models and computing thousands of simulation variables, this being the most important application in the simulator context. Through the attached SimCorba communication library, the kernel provides the client applications with services such as periodic transfer of variables, actions to control the simulation, etc. SimCorba manages all communication aspects and offers a single, easy to use programming interface to operate with the kernel. Its design has to be carefully chosen in order to allow an efficient transmission of the simulation state.

The Variable Debugger (DESI; from the Spanish ‘Depurador del Simulador’) allows the query and modification of the value of existing variables associated with the simulation models. Apart from being very useful for the validation of these models, it is widely used in training sessions for plant operators.

The Javi application (see Figure 3) is a three dimensional (3D) graphic and axial displayer of the state of the power plant core. Operators can have a global overview of the vessel and loops through graphics and color codes and so can detect problems easily. The main novelty of this application is the use of Java to provide platform independence.

The Supervisor (SPV) is another important tool that allows the selection of the simulation models from which the kernel will be composed, generating both resource and configuration files automatically. Some general simulation aspects such as cycling time can be adjusted on-line using this tool as well.
Figure 3. Screenshot of Javi application.

The **IGS Displayers** are only used in the IGS simulator context. They allow the training of operators in a classroom, differing from the simulation in the FSS simulator, where the hardware components of the control room are directly manipulated. An IGS Displayer allows the visualization of the existing control room components through graphical sheets. Students can perform actions such as opening and closing of valves, alarm recognition, etc. Furthermore, actions taken will have repercussions in the global state of the simulation session, changing conditions that will affect other students.

The **Plant Process Computer** (PPC) is an example of a subsystem developed for a specific simulator. The main goal of the PPC is the simulation of the plant process computer sited in the power plant. The PPC subsystem is composed of several physical panels, computers controlling the state of the plant (actually, each computer is responsible for a different subsystem), screens (at least 10) and keyboards associated with them. Screens display information in different formats such as alarms, bar graphs or groups of variables to report on the state of the power plant.

### 3. USED TECHNOLOGIES

This section presents a brief summary of some of the basic features of the technologies used: CCA and RT-CORBA.
3.1. CCA fundamentals

CCA provides a means for scientific software developers to build applications by assembling software components in a ‘plug & play’ environment for high-performance computing. CCA is a specification developed by the CCA Forum to describe the rules for constructing components, the model for linking them together and the collection of services that CCA-compliant frameworks should provide.

**Connection model:** Components interact with each other through well-defined ports, which are the key elements of the connection model representing communication end-points for components. A CCA port is described by an interface that declares a collection of methods without revealing implementation details. Components are linked together by connecting their ports following a provides-uses interface design pattern similar to that used in CCM. There are two types of ports: ProvidesPorts, which represent the services offered by a component and describe its calling interface, and UsesPorts, which describe the functionality a component needs. The latter are the stubs used to invoke services provided by another component. A procedure (not dataflow) relationship is established between two components when a UsesPort is connected to a compatible (same type) ProvidesPort.

**Scientific IDL:** The Scientific Interface Definition Language (SIDL) and the Babel tool [21] adopted by the CCA mean that the use of components is independent of the implementation languages. SIDL is a high-level, object-oriented, programming language-neutral Interface Description Language (IDL) used to describe component interfaces. It provides classical abstractions and data types commonly used in scientific computing, such as dynamic multi-dimensional arrays and complex numbers. Its object model has partial support for inheritance, polymorphism and method overloading. Using SIDL descriptions, Babel generates the necessary glue code to translate method calls from one language to another.

**Component frameworks:** In the context of the CCA, different frameworks have been developed to support specific computational environments such as parallel, distributed or multi-thread. Ccaffeine, the framework used in this project, is focused on local and parallel high-performance applications. It uses a trivial extension of the SPMD programming model, referred to as SCMD, where identical frameworks containing the same set of components wired the same way are instantiated in every process. Inside each process, the framework mediates component interactions through a highly efficient port mechanism implementation. On the other hand, parallel instances of the same component in different processes (referred to as a cohort) can communicate with each other through a specific parallel environment such as MPI [22], PVM [23] or Global Arrays [24].

3.2. RT-CORBA fundamentals

CORBA is a communication middleware that allows the communication among objects developed in different programming languages and running on different hosts or operating systems in a transparent way. These objects (servers) define interfaces with operations provided to the clients. The clients only use these operations in such a way that there is no difference between invocations to local objects and invocations to remote objects because all communication details are internally managed by CORBA.

Temporal predictability is a main aspect in the development of real-time applications. However, standard CORBA implementations are not suitable for real-time since they only support best-effort capacities in the communications and there is no guarantee about the temporal response on particular invocations to remote objects. So, the solution is to use ORBs supporting the RT-CORBA.
specification. RT-CORBA provides mechanisms that allow configuration and the control of processor, communication and memory resources. The following points show the main RT-CORBA features used in the implementation of the simulators.

Native and CORBA priorities: RT-CORBA applications can use CORBA priorities that allow the heterogeneity of native priorities to be hidden in the different operating systems of a distributed application. RT-CORBA priorities can be specified with values in the range 0–32,767. These priorities are used in a platform-independent way.

Server declared and Client propagated priorities: Two different policies are used to transmit priorities. In the SERVER DECLARED model, the server declares the priorities at which an invocation made on an object will be executed. The CLIENT PROPAGATED model allows the propagation of the client’s priorities that must be honored by servers, avoiding priority inversion problems [25].

Thread pools: They allow the pre-creation of threads to manage concurrent invocations. This way, the cost and unpredictability of creating dynamic threads are avoided. The thread pools can contain static and dynamic threads and can be created with lanes of different priorities, allowing the redistribution of invocations depending on client priorities.

Synchronization: RT-CORBA mutexes are the standard RT-CORBA synchronization mechanism that permits priority inheritance and priority ceiling protocols [25] if the underlying operating system supports them (e.g. POSIX).

Protocol properties: The underlying transport protocol used by a particular ORB (e.g. IIOP - TCP/IP) can be configured by RT-CORBA to benefit from special features, such as ATM virtual circuits, etc.

Connection management: Explicit binding and private connections can be used to avoid the unpredictability related to the implicit activation of objects and multiplexed connections of standard CORBA ORBs. These mechanisms permit the pre-establishing of non-multiplexed connections and control how client requests are propagated on these connections.

4. CCA-COMPLIANT SIMULATOR KERNEL

On-line applications used by operators in training sessions consume simulation data being computed and constantly updated by the simulator kernel. A delay in the simulation, owing to the low efficiency of the kernel, can not be afforded and so a high-performance component technology has been used in the development of this application. This section describes the parallel, CCA-compliant simulator kernel.

4.1. Concepts and SIDL definitions

Simulation models contain the necessary code to simulate the operation of specific power plant subsystems. However, they are not isolated pieces of code. Instead, the execution of a simulation model usually requires the reading or updating of data variables (referred to as simulation variables) which are computed by other models. Classical versions of the kernel resolved these types of inter-model data dependencies declaring all shared data as global variables and allowing access to them from any procedure. Since we pursue the encapsulation of simulation models into separated software components, we must adopt a more appropriate mechanism for managing data dependencies. In our proposal, every simulation model component must report on:
— **Provided simulation variables:** the model calculates and manages these data variables offering (exporting) their values to the rest of the models.

— **Required simulation variables:** the execution of the model involves the reading or updating of these variables, which are computed by other simulation models.

The programmer implements specific methods to declare the collections of simulation variables provided to (and needed from) the rest of the models. By calling these methods, the application can locate the requested variables and supply them to the requester components. When data dependencies involve variables hosted in different processes, specific parallel communication patterns, which are automatically established in an initial configuration phase (described later in this section), are used during the simulation. This type of automatic management of data dependencies leads to a significant uncoupling among simulation models in both development and execution time. The programmer does not need to be concerned about issues such as knowing the rest of the models in the simulator or dealing with inter-model and inter-process communications to get the requested variables and so they only need to focus on writing the scientific code of the model under development. As a result, simulation models programmed this way are easier to develop and maintain.

The use of the Babel tool makes it possible to implement the simulation models in different programming languages such as C, C++ or FORTRAN. In addition, these components can be based on sequential or parallel programming styles. The utilization of software components, the opportunity to program them in different languages and the way in which data dependencies are automatically managed, means that simulation models can be uncoupled from their contexts, which allows them to be reused in other simulators.

Simulation codes model the nuclear power plant as a mesh of interconnected nodes and cells over which variables are computed and updated at every time step. We define two SIDL classes to deal with simulation variables. Objects of these classes appear as arguments in some operations of the simulation model interface providing access to simulation data. SimReference class has a name, a node number and a range of cells. Instances of this class are used to request for specific variables. Specifically, components use lists of SimReference objects to report on simulation variables they need, for reading or updating, as well as variables they provide. The SimVariable class extends SimReference to add the specific value that the variable takes on each cell. As an example, a model can inform, through a SimReference object, that it needs to read values of ‘pressure calculated on node 2, cells from 1 to 10’ whereas the system supplies the model with these values through a SimVariable object. The following code shows the SIDL definition of the SimReference and SimVariable classes:

```idl
class SimReference {  
  void    createSimReference(in string name, in int node,  
                             in int initCell, in int finalCell);  
  string  getName();  
  int     getNode();  
  ...
}

class SimVariable extends SimReference {  
  void    createSimVariable(in string name, in int node,  
                           in int initCell, in int finalCell);  
  double  getValue(in int cell);  
  array<double> getAllValues();
}
```
Simulation models can read or modify a single cell value through the `getValue()` and `setValue()` methods, respectively. Furthermore, the `getAllValues()` operation returns an array object containing all cell values that can be directly queried or modified.

The `ISimModel` interface described below groups all the operations a simulation model component needs to implement:

```java
interface ISimModel extends gov.cca.Port {
    array<SimReference> getListRefRead();
    array<SimReference> getListRefUpdated();
    array<SimReference> getListRefProvided();
    string getModelName();
    SimVariable getVar(in SimReference reference);
    void setVar(in SimVariable variable);
    void setup();
    void initialize();
    void execute();
}
```

The `getListRefRead()`, `getListRefUpdated()` and `getListRefProvided()` return arrays of `SimReference` objects representing the simulation variables read, updated and provided by the model, respectively. A distinction between read and updated variables has to be made since the latter involves additional communications when they are hosted in different processes.

`getVar()` returns the `SimVariable` object associated with a simulation variable the model supplies. The `SimReference` object passed as argument is used to select the correct variable from those provided.

`setVar()` allows a simulation variable to put a into a model that requests it. If a `SimVariable` object is retrieved from one model and put into another, both the provider and the requester components gain access to the same simulation data.

`setup()` contains the necessary code to create the variables the model exports, as well as any other internal variables used.

`initialize()` gives initial values to the model variables. Usually, the initial state of the simulation is loaded from several configuration files that are managed by an external tool.

`execute()` contains the parallel or sequential code that implements the simulation of the corresponding power plant subsystem. Data dependencies are resolved prior to the calling of this method and so access to simulation data supplied by other models is available.

The functionality of some of these methods is independent of any considered simulation model. For this reason, the common operations are implemented in a generic base class, called `BaseModel`, from which new components can inherit. As an example, `BaseModel` implements `getVar()` and `setVar()` methods making use of efficient data containers to store and retrieve simulation variables.
4.2. Parallel kernel architecture

The architecture of the kernel consists of a central manager component, called Setru, together with a collection of simulation models connected to it. This scheme is replicated in every participant process according to the SCMD execution model. As the number of included simulation models is initially unknown and the construction of different kernels must be supported, the creation of ports to communicate with each component needs to be carried out dynamically. In the CCA model, ports can be added, removed and connected at run-time, and this is considered a normal behavior. In this sense, the CCA has an advantage over component models such as CCM that do not allow the dynamic addition or removal of ports.

Setru controls the execution of the kernel. This main component carries out a wide variety of functions such as retrieving information from the connected models, setting up data structures accessed during the simulation, resolving data dependencies, managing parallel communications to maintain data consistency, executing the simulation models or managing SimCorba to communicate with the rest of the simulator tools.

Figure 4 shows a simplified version of the simulator kernel implemented as a parallel SCMD application in the context of the Ccaffeine framework. Obviously, a real kernel usually contains hundreds of models that, in turn, can (possibly) make use of other auxiliary components.

Simulation models are classified as being either parallel or sequential according to their programming style. A component implementing a parallel model, for example, TRAC in Figure 4, uses a communication library such as MPI or PVM to divide the computation up among several processes.
aiming to achieve a reduction in execution time. Since instances of a parallel component in different processes compute different ‘parts’ of the simulation, they usually require and provide distinct sets of variables. This means that arrays returned by `getListRefRead()`, `getListRefUpdated()` and `getListRefProvided()` will depend on the process in which the method call was performed. We refer to this situation as a parallel model with distributed simulation variables.

A sequential simulation model does not split the computation or use any parallel communications at all. Instead, the same instructions are executed on every process. The utility of sequential models increases when they are able to compute many simulation variables in a short period of time, and these data are going to be read in every process by other components. In this context, it may be more efficient to provide the same variables as replicated data than to compute them in only one process, saving processing time, but resorting to parallel communications for sending and receiving values continuously.

The proposed simulator kernel supports the integration of both types of simulation models: parallel models with distributed variables and sequential models with replicated variables.

### 4.3. Execution phases

The simulator kernel execution is divided into two different phases, an initial *configuration phase* and a *simulation phase*. In the first phase, both simulation models and SimCorba are configured. The resolution of data dependencies and the creation of additional communication threads take place in this phase as well. In the second phase, the power plant simulation is carried out through the execution of the simulation models according to the commands received from client tools. Applications and tools are supplied with simulation data computed in this phase.

#### 4.3.1. Configuration phase

The structure of a specific simulator kernel, including the employed simulation models, their relative execution order and a set of global simulation parameters, is described in a configuration file. Setru uses this information to register a `ISimModel UsesPort` for each included simulation model. On the other hand, every model registers one `ISimModel ProvidesPort` to offer services to Setru, as well as any other `UsesPort` needed to call methods on auxiliary components. Figure 5 shows a reduced simulator kernel and the connections established between Setru and three simulation models. In this case, the `execute()` method on model NEMO uses functionality implemented by another component, called GSoLv. Port registration procedure takes place in the `setService()` method, which is called by the framework when the component has just been instantiated. According to the CCA specification, the implementation of `setService()` is mandatory for every component.

Once components are connected together, the following steps are carried out in parallel by Setru. First, it calls `setup()` and `initialize()` on every simulation model. These methods configure the models and prepare them for their later execution. Then, it calls `getListRefRead()`, `getListRefUpdated()` and `getListRefProvided()` to obtain information about read, updated and provided simulation variables, respectively. This local information, retrieved from components connected in every single process, is exchanged with the rest of the participant processes. From now on, all instances of Setru know the location of requested and provided variables that allows them to resolve any local and remote data dependencies in the way described below.
**Local data dependencies:** when a simulation model requires a variable computed in the same process by another model, a local data dependency occurs. Setru resolves it by calling `getVar()` on the provider component and `setVar()` on the requester. Figure 6 illustrates a scenario in which all instances of a sequential component, called Seqmod, need to read the simulation variable `sm6`, which is only provided by the part of the parallel component Parmod being executed in process 1. To resolve the local data dependency that actually happens in process 1, Setru obtains the SimVariable object associated with `sm6` calling `getVar()` on Parmod (Step 1 in Figure 6), and puts it into Seqmod calling `setVar()` (Step 2). Since SIDL objects are implemented through Java-like references.
Remote data dependencies: A remote data dependency occurs when several processes are involved. In order to resolve it, an additional (proxy) variable of the same type as the provided variable is created and managed by Setru in the process that contains the requester component. The situation is described in Figure 6. Seqmod instance in Process 2 reads the variable \( \text{sm6} \), which is hosted in process 1. This time, Setru performs actions in both processes. In process 1, Setru obtains the provided variable calling \( \text{getVar}() \) on Parmod just as it did in the previous example (Step 1 in Figure 6). In process 2, Setru creates a new SimVariable object as an intermediary proxy variable (Step 3), and passes it to Seqmod through \( \text{setVar}() \) (Step 4). During the simulation, Seqmod uses this proxy variable as if it were the real local provided variable, reading or modifying it when necessary, while Setru takes care of transferring updated values between the proxy and the real variable to maintain data consistency (Step 5). From the viewpoint of the Seqmod programmer, it is not possible to differentiate between the proxy and the remote variable. Setru hides all communication aspects and so the programmer does not need to be concerned about the location of simulation data, making component development easier.

Information retrieved from components about requested and provided variables allows Setru to establish, for each simulation model, the parallel communication pattern that consists of the minimal number of MPI messages needed to resolve its remote data dependencies. These communication patterns, automatically calculated in the configuration phase, remain unchanged during the rest of the simulation, so that an efficient message passing scheme can be achieved.

4.3.2. Simulation phase

This phase mainly involves the execution of the simulation models following a specific order, initially described in the configuration file. The group of simulation models are executed sequentially, one after another, whereas each model runs in parallel through several processes. The execution of a single model entails the following three steps that are carried out by Setru.

1. **Proxies updating**: proxy variables are updated with the values of requested simulation variables that are hosted in other processes.
2. **Execution**: the \( \text{execute}() \) method of the component is invoked in every process.
3. **Variables returning**: values of updated proxy variables are returned to the processes containing the original simulation variables in order to modify them.

Parallel communications used to resolve data dependencies occur in both Step 1 and Step 3. They are performed using the following two-stage communication protocol: first, each Setru instance sends data to the processes requesting the variables it owns, and second, it receives data from the processes providing the variables it requires. In the worst case, each process may need data from the others in an ‘all-to-all’ communication scenario that may cause deadlocks. These types of deadlocks, however, are easily avoided by using non-blocking communication primitives for the initial sending operations. When several simulation variables are going to be transferred from one process to another, they are previously packed together aiming to minimize message passing.

In this phase, the kernel also reacts to different simulation commands such as \textit{stop simulation}, \textit{debugging mode}, \textit{load initial conditions}, etc. that are emitted by the client applications.
As the main thread of every process is in charge of executing the models, communications between the kernel and client tools are supported by additional threads through SimCorba (as described in Section 5).

Obviously, the use of software components, together with a generic runtime system that supports the construction of different kernels can lead to a certain loss of performance when compared to a specific classical (non-component-oriented) version programmed in parallel as well. This overhead can be reduced when reasonable amounts of computation take place in the models, which is usual for the described simulators. In recent work [26], we presented a prototype implementation of this CCA-compliant power plant simulator kernel. In order to measure the above mentioned loss of performance, two simulation scenarios were considered: computationally intensive and communication intensive with lots of remote data dependencies. The results showed that the penalty overhead (in execution time) was lower than 5% in all parallel and sequential experiments carried out. This shortcoming can be afforded as it is compensated for by the many advantages gained from the application of component oriented paradigm to this area.

5. COMMUNICATION ISSUES

Interactions between the kernel and client applications are mainly performed by means of simulation variables, similar to those used in the simulation models. So, for example, digital variables can be used to represent devices such as keyboard input or lights in physical panels. Inside the simulator kernel, we established a mechanism to exchange variables among simulation models in an efficient way. Unfortunately, the use of such a specific mechanism is limited to that parallel kernel and so other strategies must be adopted for intensive communications that take place among the distributed tools from which the simulator is composed. In addition, some of these applications require a predictable temporal behavior as they have soft real-time constraints: the refreshing of the screens, the response time of user actions and so on. We have used RT-CORBA for these purposes. Specifically, we have chosen TAO [27], a freely available CORBA ORB very suitable for real-time applications due to features such as predictable timing and robustness.

Two types of communications take place in the simulator. On the one hand, the simulator kernel has to respond to simulation command requests from client applications (sporadic actions): start simulation, stop simulation, load initial condition, etc. On the other hand, the simulator kernel must periodically provide simulation variables to the rest of the applications.

5.1. Communication architecture

As described in Section 4, simulation models are executed and controlled by the Setru manager component in each process of the parallel simulator kernel. This architecture influences the communication design and so we have one CORBA object (server) hosted in each of these parallel processes. These objects, which offer to client applications a suitable CORBA interface to perform both command reception and simulation variable transferring, are created and managed by the Setru component through the SimCorba communication library. SimCorba ‘subcomponents’ have their counterpart in the client side (simulator tools) in such a way that each application that wants to interact with the kernel manages its own client communication component, so-called Receiver. Figure 7 shows this situation.
Figure 7. SimCorba and Receiver communication components.

The Receiver component was designed aiming to achieve both reutilization and hiding communication codes from applications. From the client’s point of view, the behavior of the Receiver component acts as a passive data container with arrays that can be consulted and automatically updated with simulation data. Furthermore, this component offers a set of additional services, which allows the sending of commands to the simulator kernel as well as modifying its simulation variables.

The interaction between SimCorba and Receiver components is performed following a client-server style. SimCorba is activated in every parallel process of the simulator kernel waiting for invocations from client applications. When an application wants to receive data or to send commands, it initializes a Receiver component as well.

A Receiver component needs to obtain the reference of a SimCorba object before using it. Therefore, the Naming Service [28] of CORBA, in which distributed objects can be registered and subsequently located through names, is employed. This way, Receiver components use names to obtain the references of previously registered SimCorba remote objects.
5.2. Command services

SimCorba offers different services, which allow the sending of commands to the simulator kernel. The following code shows the CORBA IDL definitions of some of these services:

```idl
interface ISimulator {
    // start-stop methods
    boolean send_run(out string errMsg);
    boolean send_freeze(out string errMsg);
    boolean get_state(out t_Sim_State state, out string errMsg);

    // simulation control services
    boolean send_slowtime(in long slow, out string errMsg);
    boolean send_normal_time(out string errMsg);
    boolean send_fast_time(in long fast, out string errMsg);
    boolean send_compute_n(in long nsteps, out string errMsg);
    boolean send_step_n(in long n_step, out string errMsg);
    boolean send_step_cont(out string errMsg);

    // initial condition services
    boolean send_load_ic(in short icnumber, out string errMsg);
    boolean send_save_ic(in short icnumber, out string errMsg);
    boolean send_backtrack(in long num, out string errMsg);
    ...
};
```

There are three categories of methods.

— Start-stop methods allow the starting or stopping of the simulation. So, for example, to start the simulation, a Receiver component performs a `send_run()` invocation.
— Control services manage the simulation behavior. So, for example, the service `send_step_n()` allows the execution of the simulation running \( n \) steps and then the pausing of the simulation.
— The last services allow the handling of all the initial conditions in the training sessions: loading, saving, etc. These initial conditions store the values of all simulation variables so that they can be used in later sessions.

In general, when a Receiver component wants to invoke a simulation command service, it has to repeat the invocation in every SimCorba object. It can be argued that synchronization is a problem or that this strategy is not efficient. However, synchronization is the responsibility of the simulator kernel and neither SimCorba nor Receiver take care of it. When a SimCorba object receives an invocation, it delegates the responsibility of the command to methods executed in its corresponding parallel process of the simulator kernel, which is responsible for synchronizing with the other processes. As regards performance, another alternative is to have only one single SimCorba object in the simulator kernel (in one process only) in order to avoid the ‘one-to-many’ calls carried out by Receiver. However, this approach implies losing the possibility of communicating directly with the other processes of the simulator kernel, which is necessary to query variables in an efficient way as explained in the following section. Taking into account that transferring simulation variables is the most important action for client applications, we must maintain a SimCorba object per process.
5.3. Simulation variable services

The management of simulation variables includes actions such as queries and modifications. As described in Section 4, there are simulation variables that are contained in all processes (replicated), whereas other variables are provided only by specific processes (distributed). So, when a Receiver wants to query the value of some variable, it needs to select the right SimCorba object to perform the invocation. The way to know the correct SimCorba is through initialization. When a Receiver component is initialized, it receives lists with the variables that every kernel parallel process provides. Receiver components only have to use this information to choose the right SimCorba. For example, the IGS application in Figure 7 is accessing to simulation data that are being computed (and hosted) in Processes 1 and 2. It is important to remark that these steps are transparent to the developer of the application.

The following code shows some of the CORBA IDL definitions related to variable management:

```c
// single variables
struct analogVble {
    float value;
    long id;
};

struct digitalVble {
    unsigned short value;
    long id;
};

struct inputVble {
    string name;
    double value;
};

// lists of variables
typedef sequence<analogVble> listAnalogVbles;
typedef sequence<digitalVble> listDigitalVbles;
typedef sequence<float> secAnalogValues;
typedef sequence<unsigned short> secDigitalValues;
typedef sequence<inputVble> secInputVbles;

// all variables
struct StructAllValues {
    secAnalogValues analogues;
    secDigitalValues digitals;
};

// changed variables
struct StructChangedValues {
    listAnalogVbles analogues;
    listDigitalVbles digitals;
};
```
interface ISimulator {
    exception NoSession {};  
    exception UndefinedType {}; 

    // variable requests 
    StructAllValues sendAllValues(in string type) 
    raises (NoSession, UndefinedType); 
    StructChangedValues sendChangedValues(in string type) 
    raises (NoSession, UndefinedType); 

    oneway void writeValues(in secInputVbles s); 

    void notifyAllValues(); 
    void notifyChanges(); 

    // single variable request 
    boolean send_var_query(in string varName, inout char varType, 
                            in short cell, in short comp, out double value, ...); 
    ...
};

There are several IDL structs with definitions for the simulator data similar to those contained 
in the SIDL interface of the simulator kernel (Section 4). In this case, the definitions are 
optimized for the communications with client tools. So, we have definitions for analogue and digital 
variables (analogVble, digitalVble) or sequences (similar to lists) of analogue and digital variables 
(listAnalogVbles, listDigitalVbles). 

The interface ISimulator contains several methods to obtain simulation variables. The collection of 
data sent by SimCorba is closely related to the client type. This type is determined by the application 
that uses the Receiver component. SimCorba has the necessary information to know the variables 
required for each type in a flexible way, allowing the incorporation of new client types without having 
to modify the application code. 

SimCorba and Receiver have to deal with a huge number of variables (about 26000), which must 
be updated at a range of four times per second. There are two alternatives for tackling this issue: 
transferring all the variables in every updating procedure (using arrays) or only the changed variables 
(using CORBA sequences). The first alternative has advantages such as an easier C++ mapping for 
programmers or avoiding problems such as lost changes. On the other hand, the huge data volume 
makes it advisable to use the second alternative. After carrying out performance tests, the use of 
CORBA sequences with only changed variables is chosen to the detriment of transferring complete 
variable arrays. Nevertheless, both ways of data transferring are available and the applications can 
use either of them. So, for example, if a client tool such as PPC wants to retrieve all its variables, it 
will invoke the sendAllValues() operation, receiving a struct of type StructAllValues that will 
contain all the values of the PPC type in CORBA sequences. If the application only wants to recover 
the changed values, it will invoke sendChangedValues(). In addition, single variables can also 
be queried. In this case, the client can use the operation send_var_query(). It is the responsibility 
of the Receiver component to invoke these operations on the correct SimCorba objects. On the other 
hand, SimCorba has to maintain updated lists of changed variables in the simulator kernel.
This interface also allows user actions such as keystrokes. The writeValues() operation is used for this purpose. In this operation, the client provides a list of user actions that consists of name-value pairs (struct inputVble) as arguments. User actions are performed updating the variables in the simulator kernel, which affects other parts of the simulation.

The Receiver component is implemented on dynamic libraries where all communication details are hidden, offering a single API with arrays, which can be easily manipulated by the clients. The following code shows a fragment of this API implemented in the CReceiver class, used by C++ client applications:

```cpp
class CReceiver {
public:
    void initialize(char *Receiver_type, ...);
    bool thereisSession();
    void requestAllValues();
    float *getAnalogueVariables(long &num);
    unsigned char *getDigitalVariables(long &num);
    void setAnalogueVariables(float *vars, long num);
    void setDigitalVariables(unsigned char *vars, long num);
    void setValues(char *pvalues_list);
    ...
};
```

The first call the client must carry out is initialize(), indicating the application type. In this initialization, several tasks such as connecting to the Naming Service or obtaining references to the SimCorba objects are performed. Once this method has finished, the client has two arrays at its disposal: an analogue array and a digital array. Arrays are manipulated as with any other C/C++ array through pointers: float * for the analogue array and unsigned char * for the digital array. All the variables needed by the different types of client tools are contained in these arrays and, in addition, the clients will have information about which variables are in each position of the arrays. The clients do not need to know anything about SimCorba. Once again, communication details are hidden and client applications have only to deal with common C/C++ arrays, internally updated from SimCorba.

As an example, the following code shows how to obtain the analogue variable number 0, which represents the Simulation Time.

```cpp
CReceiver theReceiver;
float *analog_array;
long num_anlgs;

theReceiver.initialize(...); analog_array =
theReceiver.getAnalogueVariables(num_anlgs); cout << "Actual
time:"<< analog_array[0] << endl;
```

### 5.4. Applying real-time features

In SimCorba, there are thread pools for client invocations. With these thread pools, we can guarantee processor resources for these invocations. In addition, we have used the client propagated model of
RT-CORBA in which the method invocations are executed taking into account the priority of the caller. This way, we can associate different priorities with the client applications. So, for example, since Instructor Console commands are more important than DESI (variable debugger) commands, the former has a higher priority than the latter, aiming to improve the temporal behavior of the most important applications (see Figure 8).

The number of static clients is known when SimCorba is started, which allows for it to create thread pools with lanes for the static clients and for possible dynamic clients. The thread pools are configured according to the priorities of the different client types. On the other hand, when the Receiver components are initialized, the connection with SimCorba is verified using the explicit RT-CORBA binding mechanism. This way, the Receiver components will have reserved suitable network resources. Private connections with priority banding are also used when possible to guarantee QoS between SimCorba and Receivers.

Synchronization is a fundamental aspect in multi-thread applications. In this sense, there are mutual exclusion zones in different parts of SimCorba, which have been protected through RT-CORBA mutexes. So, when an application invokes a method that needs to access one of these zones, the mutex avoids conflicts in the usage of that part of the kernel and so, for example, two different clients can not simultaneously execute contradictory commands, such as start or stop the simulation. RT-CORBA mutexes respect the priority of the invocations avoiding priority inversion problems.

Finally, timeouts on invocations have been used in the applications, which have control over the temporal response of SimCorba and can report to the user about hypothetical problems.

6. CONCLUSIONS

Nuclear power plant simulators have been traditionally developed using ‘ad hoc’ solutions based on languages such as C, FORTRAN or ADA. However, classical programming techniques are not oriented
to improving evolution, reusability or maintenance of large software systems, valuable aspects for a simulator that suffers from dynamic changes constantly: addition and modification of simulation models, applications and subsystems, version management, different programming teams, etc.

The first motivation for using component-based technologies was to overcome these shortcomings, thereby improving the simulator software life cycle. Our system comprises both a simulator kernel and a group of related applications. The former computes the power plant simulation by executing the simulation models (scientific codes), whereas the latter consists of a collection of client applications that interact with the kernel sending commands and receiving simulation variables. From the beginning, we took into account the different nature of these two scenarios, which led us to combine two different component technologies.

High performance has become the main requirement for the simulator kernel. Since the majority of computationally intensive simulation models of our system can be parallelized to reduce their execution time, the component model on which the kernel should be based had to provide parallel execution and efficient interactions among software components. The high-performance computing oriented CCA offers these features. The SCMD execution model provided by the Ccaffeine framework matches perfectly the way our kernel is componentized and parallelized following a SPMD style. Thus sequential and parallel simulation models are encapsulated into software components that implement a common interface. A generic architecture based on a single manager component (called Setru) to which the simulation models are connected, allows the construction of different kernels adapted to different scenarios, as well as promoting the reusability of scientific code. In order to uncouple simulation models as much as possible, they declare both required and provided simulation variables, so that Setru is in charge of resolving all inter-model data dependencies automatically. This is achieved through an effective algorithm that uses minimal parallel communications to transfer the required data among processes, and releases the programmer from this tedious task, which involves the precise knowledge of the rest of the simulator or the establishment of communications to access distributed data. This considerably reduces the development complexity and imposes a standard for dealing with simulation data to the programmers.

Requirements are different for other parts of the simulator. In this case, we need to achieve efficient communications among distributed applications with soft real-time constraints, developed in different programming languages and executed on different operating systems or hardware platforms. RT-CORBA provides a suitable solution for this purpose. Communications involve interactions between client applications and the simulator kernel for executing simulation commands or intensive transferring of simulation data. We have developed a reusable communication infrastructure that allows the inclusion of new tools with minor modifications in the simulator. SimCorba and Receiver are CORBA-based communication components that belong to the simulator kernel and client applications, respectively. They hide all communication details from both the simulation model and the client tool developers, offering the latter an easy to use programming interface to operate the kernel. SimCorba and Receiver implement the suitable mechanisms to transfer simulation variables to the client tools periodically, taking advantage of RT-CORBA real-time features such as thread pools, execution priorities or resource management.

We have obtained many benefits from the use of software components to model the simulator. Reduction of complexity in the development process of both simulation models and client applications was achieved, as well as many other software maintenance related aspects being improved. However, we think that the most important of these advantages will be strongly manifested in future stages of the
simulator software life cycle. Apart from nuclear power plants, we are convinced that similar design patterns, based on CCA and RT-CORBA, can be followed to build simulators in very different contexts.

ACKNOWLEDGEMENTS

We would like to acknowledge the help from the staff of the Simulation Division at Tecnatom S.A. who contributed to the preparation of this paper. In addition, we are very grateful to the referees whose critical comments have improved the paper.

REFERENCES


