SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

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, E.T.S. Ingeniería Informática, University of Málaga, 29071 SPAIN

SUMMARY

This paper proposes the application of modern component-based 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 comprising the simulator, improve the evolution and maintenance of the entire system, as well as promoting code reusability in other projects.

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

1. INTRODUCTION

Simulators are specially important in the context of nuclear power plants since they can predict the plant status when facing different situations that can occur in the daily operations. In this

*Correspondence to: Sergio Romero, Department of Languages and Computing Science, E.T.S. Ingeniería Informática, University of Málaga, 29071 SPAIN
†E-mail: sromero@lcc.uma.es

sense, a fast response is required and performance becomes a major factor. Besides that, they can be used as training tools for future operators, allowing the practise 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 basic working is simple. 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 and purposes comprise the simulator software architecture. In this context, the most important application is the *simulator kernel*, which is responsible for executing lots of scientific codes implementing detailed mathematical models of the different power plant physical subsystems. There is a wide range of these *simulation models*, from computationally intensive complex models like TRAC (thermo hydraulic model) or NEMO (neutronic model) to simpler ones simulating, for example, the operation of a valve.

In previous work, we have collaborated with the company Tecnatom S.A. [23] in the development and maintenance of different simulators currently used in several power plants located in Spain, Germany and Mexico [5] [6]. The kernel of these simulators was programmed in a classical style, in such a way that all simulation models, coded as sequential Fortran and C procedures, were statically linked together into the kernel. Although feasible at first, this approach 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 [11]. Based on component interoperability, this programming style allows the creation of more flexible and adaptable software. Nuclear power plant simulators constitute large complex software systems that can take advantage of componentization in order to improve evolution, maintenance and software life cycle. However, parallelism and high performance, which are requirements of fundamental importance for a simulator kernel that needs to execute complex mathematical models in acceptable time, are not taken into account by component standards and implementations such as OMG CCM [20], Microsoft DCOM [13], Sun Java Beans and Enterprise Java Beans [9] [16]. 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, traditionally based on classical programming techniques and languages such as FORTRAN and, more recently, Java and C++ [26]. In this sense, ASSIST [25] is focused on high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software. SBASCO [8] 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 [24] 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 high performance computing-oriented component architecture proposed by CCA and the OMG Real-time CORBA extension (RT-CORBA) [21].

We focus on the idea that a component-based simulator kernel can solve many software maintenance related problems which 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 which implements the kernel runtime system that is in charge of executing them properly.

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 by splitting the computation in such a way that they could be run on multiple processors aiming to reduce their execution time. For example, in [2] 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 with the Single Component Multiple Data (SCMD) parallel execution model provided by the CCA-compliant Ccaffeine framework [1]. 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 making, as a consequence, 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) [19] 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

Figure 1. Details of Full Scope Simulator (left) and Interactive Graphic Simulator (right)

used the Real-time CORBA specification (RT-CORBA). Other interesting approaches for developing distributed real-time simulators are based on the TMO model [14] and HLA [27].

The rest of the paper is structured as follows. An overview of the simulator architecture is provided in section 2. Section 3 presents some of the basic features of used technologies. In section 4 both the CCA-compliant simulator kernel and the model it is based on are described. Implementation decisions about employed communication mechanisms are discussed in section 5. Section 6 presents some of the most outstanding tools and applications in the simulator context. 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 (see figure 1, left).
- *Interactive Graphic Simulator (IGS)* allows operator training through graphic applications (see figure 1, right).

The following describes some of the main high level hardware elements of FSS and IGS:

- *Simulation Computers* are responsible for the simulation process executing the simulation models and providing data to the rest of software and hardware components.
- Depending on the power plant, there will be additional *Hardware Subsystems* included in the simulator.

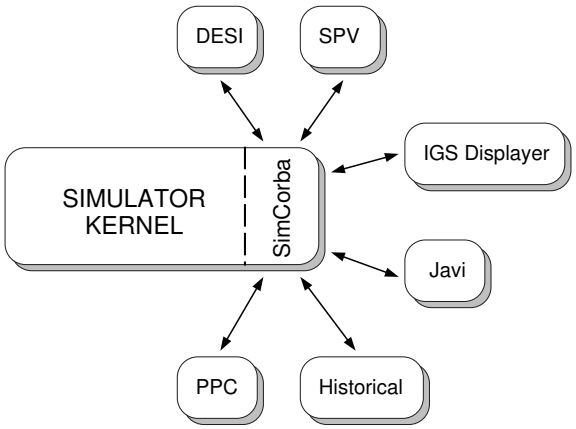

Figure 2. High level software applications conforming the simulator

- The *Instructor Console* 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 that can be executed on different nodes of a network, interacting with each other through the high-level communications mechanisms of CORBA. 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, CORBA-based 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. The first one, comprising the simulator kernel and SimCorba, acts as a simulation server offering a set of services to the rest of the client applications and tools, which constitute the second part of the system. Some of the most important high level applications together with their interactions can be seen in figure 2. The following is a brief description of their functionality:

Simulator Kernel: To compute the simulation of the power plant, the parallel simulator kernel executes simulation models and calculates lots of simulation data, this being the most important application in the simulator context. Through an attached CORBA-based communication layer, referred to as SimCorba, the kernel offers a set of services such as periodic

transfer of variables, actions carried out on the simulator, etc. to the rest of applications. The design of SimCorba has to be carefully chosen. Its main role consists of connecting the high performance parallel simulator kernel with the client distributed application environment in an efficient way, managing all communication aspects and offering a single, easy to use programming interface to operate with the kernel.

Client applications: They are a wide group of on-line applications that communicate with the simulator kernel (through SimCorba) for many different purposes, such as debugging the simulation process, allowing representation and modification of simulation variables, changing simulation aspects like cycling time, recording or restoring the simulation state in real-time, making elaborate graphical and printed reports of the power plant status, etc.

Due to 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.

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. According to it, there are two types of ports: *provides ports*, which represent the services offered by a component and describe its calling interface, and *uses ports*, which describe the functionality a component needs and are the stubs used to invoke services provided by another component. A uses port can be attached to a compatible (same type) provides port of another component. From the moment when this connection is performed, a procedural (not dataflow) relationship is established and any functionality represented by the uses port is available by invoking the methods in the connected provides port.

Scientific IDL: The Scientific Interface Definition Language (SIDL) and the Babel tool [4] adopted by CCA mean that the use of components is independent of the implementation languages. SIDL is a high level, object oriented, programming language-neutral IDL used to describe component interfaces. It provides classical abstractions and data types commonly used in scientific computing, such as dynamic multidimensional arrays and complex numbers besides

other useful features like enumerated types, symbol versioning and name space management. 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: A framework represents a concrete implementation of the model mechanisms. Component instances are created and managed within a framework that must provide, according to the CCA specification, a minimal set of communication services to the components. In the context of CCA, different frameworks have been developed to support specific computational environments such as parallel, distributed or multithread. Ccaffeine, the framework used in this project, is focused on local and parallel high performance applications. Single Program Multiple Data (SPMD) is certainly the most widely used style of parallel computing, where all processes run the same program, although each one has its own data. Ccaffeine uses a trivial extension of this paradigm, referred to as Single Component Multiple Data (SCMD), where identical frameworks containing the same set of components wired the same way are instantiated in every process. Inside each process, framework mediates component interactions through a highly efficient port mechanism implementation. Since all components are loaded into the same address space (process), when a component needs to connect a uses port to the provides port of another component to call a method, the framework returns a direct reference (pointer) to the actual implementation, which causes a minimal latency overhead for component interactions equivalent to a C++ virtual function call. 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 concrete parallel environment such as MPI [22], PVM [10] or Global Arrays [17]. CCA and Ccaffeine make it possible to construct high performance applications by connecting parallel components that are (possibly) implemented in different programming languages and even make use of distinct parallel communication libraries.

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 ones 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 Real-time CORBA specification. Real-time CORBA provides mechanisms that allow configuration and 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: Real-time 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 ranging from 0 to 32767. 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 [3].

Thread pools: They allow the pre-creation of threads in such a way that a thread manages each invocation on a particular object. This way, the cost and unpredictability of 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 [3] 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

As stated in section 2, the power plant simulator comprises a collection of applications running in a distributed environment. The simulator kernel, unlike the rest of the applications, demands higher levels of performance to run large amounts of scientific code efficiently. It calculates lots of simulation data that have to be subsequently supplied to the rest of client tools and applications. Due to this performance requirement, we have chosen CCA and Ccaffeine framework as the component technology to develop this application. This section describes the new parallel, component-oriented, 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 reading or updating data variables (referred to as *simulation variables*) which are computed by other models. Previous (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, each 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.

According to this, the programmer of a specific simulation model only has to declare, implementing the corresponding methods of the component interface, the simulation variables provided to (and needed from) the rest of the models, whereas the runtime system, and not the programmer, is the one responsible for locating the requested variables and supplying them to the respective models. 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, he/she only needs to be focussed on writing the scientific code of the model under development. As a result, simulation models programmed this way are easier to develop and maintain.

By using a tool for language interoperability like Babel, simulation models can be coded in different high performance languages such as C, C++ or FORTRAN. Since they are all CCA-compliant components implementing a specific interface, we make sure they can be integrated into the kernel. The development of these components can be based on both sequential or parallel programming styles with different communication libraries such as MPI or PVM. 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. Certainly, one of the advantages obtained from the kernel componentization lies in the possibility of constructing different kernels, adapted to different power plants, by selecting and composing the corresponding simulation models from a component repository.

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 in order to work with simulation variables. Objects of these classes appear as arguments in some operations of the simulation model component 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. In fact, components use lists of SimReference objects to report on simulation variables they need, for reading or updating, as well as variables they provide. 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 provides these values through a SimVariable object. The following code shows the SIDL definition of the SimReference and SimVariable classes:

```
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();
  void          setValue(in int cell, in double value);
  void          setAllValues(in array<double> values);
  void          assign(in SimVariable variable);
  SimVariable   subVar(in int initCell, in int finalCell);
  string        toString();
}
```

Simulation models can read or modify a single cell value through the `getValue()` and `setValue()` methods. However, in order to allow higher performance on accessing simulation data, `getAllValues()` returns an array object containing all cell values which can be directly queried or modified.

The ISimModel interface described below groups all the operations a component has to implement in order to be included into the kernel as a simulation model.

```
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 following describes the functionality of the main methods:

`getListRefRead()`, `getListRefUpdated()` and `getListRefProvided()`: These methods return arrays of SimReference objects representing the simulation variables read, updated and

provided by the model respectively. This information, which is obtained from every connected model, determines for any considered simulation variable both the model that supplies it and the ones that request it, allowing the runtime system to resolve data dependencies. A particular distinction between read and updated variables has to be made since the latter involves additional communications when they are hosted in different processes.

`getVar()` and `setVar()`: The former 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 the provided ones. Through the `setVar()` method, the runtime system can put the required simulation variables into the model. By putting a previously obtained SimVariable object into a model that requests it, both the provider and the requester models gain access to the same simulation data.

`setup()` and `initialize()`: The first one contains the necessary code to create the collection of variables the model exports as well as any other internal variables used. The arrays of SimReference objects returned by the above described methods are also configured here. The code of `initialize()` gives initial values to the model variables. Usually, the initial state of the simulation is loaded from several configuration files that can be easily created and managed by an external tool.

`execute()`: This method contains the parallel or sequential scientific code implementing the simulation of the corresponding power plant subsystem. Since data dependencies are resolved prior to the calling of `execute()`, access to simulation data supplied by other models is available. This method is called iteratively during the simulation.

A brief review of the functionality of these methods shows that some of them are independent of any considered simulation model. For this reason, we provide a new class called BaseModel that offers an implementation of these common methods facilitating, this way, the development of new components. Taking advantage of the mechanisms provided by SIDL and Babel, classes that implement the ISimModel interface can (optionally) inherit from the above mentioned base class, even if they are going to be programmed in different languages. BaseModel implements the `setVar()` and `getVar()` methods making use of efficient data containers to store and retrieve SimVariable objects. It also offers "set" methods to establish both the arrays of SimReference objects and the name of the model, implementing the operations that return them as well. By inheriting BaseModel, the programmer only needs to code the specific `setup()`, `initialize()` and `execute()` methods of the simulation model under development. In SIDL, new simulation models are defined as subclasses of BaseModel as follows:

```
class Trac extends BaseModel implements-all
      ISimModel, gov.cca.Component {
}
```

4.2. Parallel kernel architecture

Different simulator kernels share the same software architecture consisting of a central manager component, so-called Setru, together with a collection of simulation models connected to it. This scheme is replicated in every participant process according to the Single Component Multiple Data execution model. As the number of included simulation models is initially

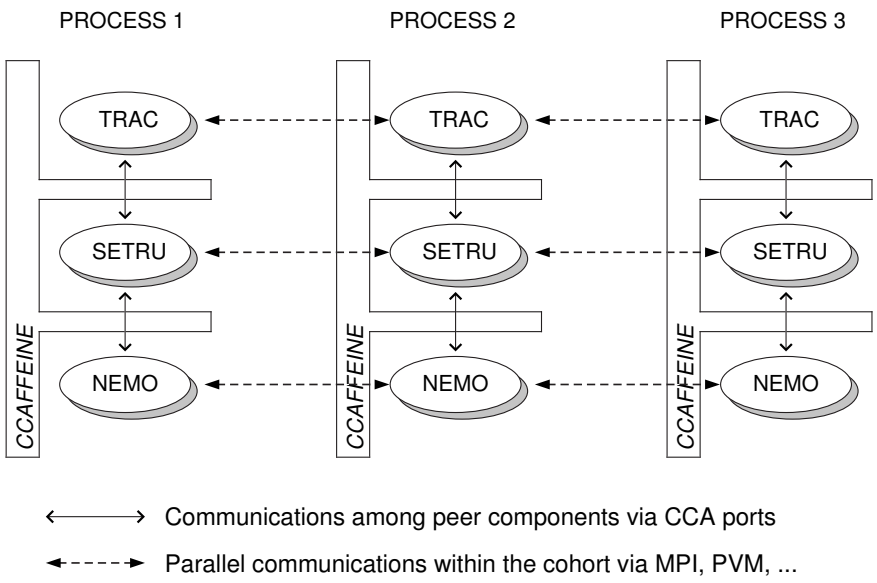

Figure 3. Single Component Multiple Data parallel simulator kernel

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, CCA has an advantage over component models such as CCM that do not allow the dynamic addition or removal of ports. CCM connections are considered part of application assembly, and the end user has nothing to do at run-time.

Setru takes care of controlling the simulation executing the connected models in the proper way. In fact, this component plays a major role since it implements the entire runtime system of the kernel, carrying 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, or managing SimCorba to communicate with the rest of on-line tools in the simulator. In the described kernel, Setru has been programmed as a parallel component using C++ and MPI.

Figure 3 shows a simplified version of the simulator kernel implemented as a parallel, SCMD application using the Ccaffeine framework. Obviously, a real kernel usually contains hundreds of models which, in turn, can (possibly) make use of other auxiliary components. In every process, both the framework and the same collection of components are instantiated. Interactions in the same address space (process) are carried out through efficient CCA ports. This occurs, for example, when Setru calls methods on the connected models.

Simulation models are classified as being either parallel or sequential according to their programming style. A component implementing a parallel model, e.g. TRAC in figure 3, 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()` methods will depend on the process the method call was done in. We refer to this situation as a parallel model with distributed simulation variables. As a very useful particular case, a model can be easily programmed in such a way that the entire computation is carried out in only one process, thereby releasing the rest of them from executing any code.

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 model is instantiated in. Utility of sequential models grows 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 these contexts, 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. In any case, the proposed simulator kernel supports the integration of all these different types of simulation models: parallel models with distributed variables, models executed in only one process, 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 fist one, both simulation models and SimCorba are configured. Resolution of data dependencies and creation of additional communication threads take place in this phase as well. In the second one, 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 provided 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. With the developed components being stored in a repository, the construction of different kernels can be easily carried out by changing the contents of this file. Setru uses this information to register a ISimModel uses port for each included simulation model. On the other hand, every model registers one ISimModel provides port to offer services to Setru, as well as any other uses port needed to use functionality of auxiliary components. Figure 4 shows a reduced simulator kernel and the connections established between Setru and several simulation models. In this case, `execute()` method on model NEMO uses functionality implemented by another component, the so-called GSolv. Port registration

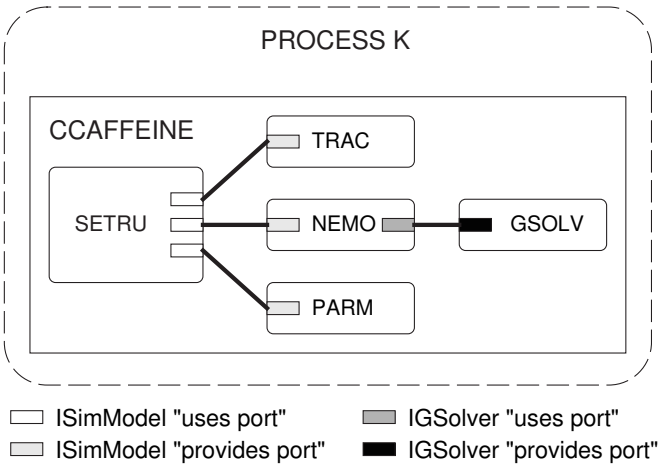

Figure 4. Simulation models connected to the Setru manager component

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 using parallel communications. From now on, all instances of Setru know the location of requested and provided variables which 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, entailing all component operations in the same process. Figure 5 illustrates a scenario in which all instances of a sequential component, the so-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 one. To resolve the local data dependency that actually happens in process one, aiming to give Seqmod access to the variable, Setru obtains the SimVariable object associated with `sm6` calling `getVar()` on Parmod (1), and puts it into Seqmod calling `setVar()` (2). Since SIDL objects are implemented through Java-like references (or pointers) by the respective

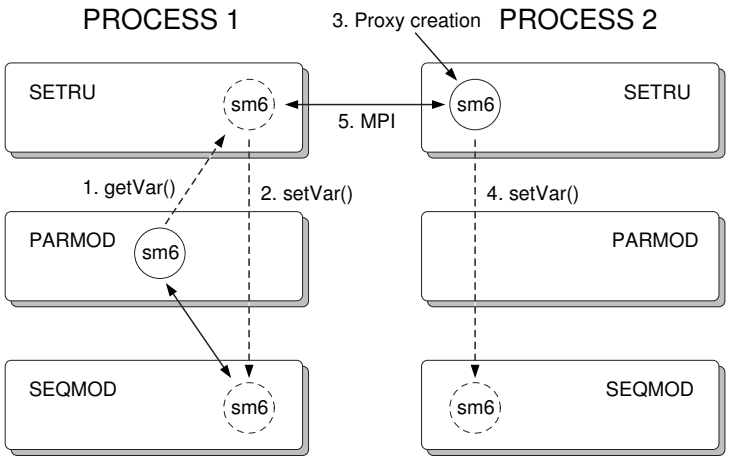

Figure 5. Resolving local and remote data dependencies

programming languages, both Seqmod and Parmod components are really making use of a single SimVariable instance referenced from two different points.

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 one is created and managed by Setru in the process that contains the requester component. The situation is described in figure 5 again. Seqmod instance in process two reads the variable `sm6` which is hosted in a different process. This time, Setru performs actions in both processes. On the one hand, Setru in process one obtains the provided variable calling `getVar()` on Parmod just like it did in the previous example (1). On the other hand, Setru in process two creates a new SimVariable object as an intermediary proxy variable (3), and passes it to Seqmod through `setVar()` (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 variable and the real one to maintain data consistency (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 related to resolving data dependencies and so, the programmer does not need to be concerned about the location of simulation data, making component development easier.

Remote data dependencies require parallel communications to supply the models with the latest computed values. 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

In this phase, the kernel can react to different simulation commands such as *start simulation*, *debugging mode*, *execute n steps*, *save simulation*, etc. which are emitted by the client applications. Once the proper command is received, the simulation begins with the execution of the models following a specific order initially described in the configuration file. According to it, the group of simulation models are executed sequentially, one after another, whereas each one runs in parallel through several processes. Since the main thread in 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). The execution of a single simulation model entails the following three steps that are carried out by Setru:

1. Proxy variables are updated with the values of requested simulation variables hosted in other processes.
2. The component is executed by calling its `execute()` method in every process.
3. Once the model computation finishes, 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 one and step three. They are carried out 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. As an extreme situation, each process may need data from the others in an "all-to-all" communication scenario that may cause deadlocks in the case where every process were to become blocked waiting for ending their `send` operations and, at the same time, with none of them executing the corresponding matching `receive`. Due to this fact, initial sending operations are implemented through MPI nonblocking communication primitives, avoiding any type of deadlocks and delays. This way, we make sure that all processes reach the `receive` operation having previously sent their data. Values of different simulation variables that are going to be transferred from one process to another are previously packed together aiming to minimize message passing.

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 (not 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 [7], we presented a prototype implementation of this CCA-compliant power plant simulator kernel. In order to measure the above mentioned loss of performance, different simulation environments such as scenarios which are computationally intensive (or communication intensive having lots of remote data dependencies) were taken into account in several tests. The results showed that the penalty overhead (in execution time) imposed by the CCA based implementation was lower than 5% in all parallel and sequential experiments carried out. Anyway, we can afford

this shortcoming as it is compensated for by the many advantages gained from the application of component-oriented paradigm to this area.

5. COMMUNICATION ISSUES

The CCA-compliant simulator kernel implements an efficient simulation infrastructure, easy to modify and reuse. However, the simulator includes many other tools such as physical panels, debuggers, keyboards, screens, etc. which support the kernel and communicate with it to perform different tasks.

Interactions between the kernel and these client applications are mainly performed by means of simulation variables, similar to the ones 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 application and so, other strategies must be adopted for intensive communications that take place among the distributed tools which the simulator is composed of.

These simulator tools can be programmed in many different languages: C++, Java, Ada, etc. CORBA is a suitable communication middleware for this situation, because it allows the communication of applications not only developed in different programming languages, but also running in different operating systems and hardware platforms. In addition, simulators impose soft real-time constraints and standard CORBA ORBs only support best-effort capacities. There is no guarantee about temporal behavior and the response time of remote invocations is not bounded. So, we have used a real-time extension of CORBA (RT-CORBA) for this purpose. Specifically, we have used TAO [15], a freely available CORBA ORB very suitable for real-time applications due to features like 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 provide simulation variables to the rest of the applications periodically. The simulator kernel offers an easy to use common interface for both types of services.

5.1. Communication architecture

CORBA applications are based on objects. The abstraction for a CORBA-based application consists of several objects that communicate among themselves, independently of their location.

As described in section 4, several 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 sending, are created and managed by the Setru component through a small communication library called

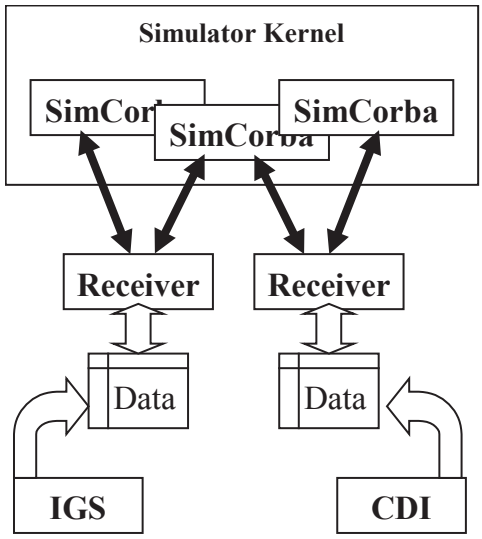

Figure 6. SimCorba and Receiver communication components

SimCorba (name derived from "Simulator CORBA"). SimCorba "subcomponents" have their counterpart in the client side (simulator tools) in such a way that each application that wants to communicate with the kernel manages its own client communication component, so-called Receiver. Figure 6 shows this situation.

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 is like 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 any 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 objects before using it. According to it, the Naming Service [12] 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 (actually, the references of all SimCorba objects, one per process, are needed). Once references are retrieved, the invocation of services on the simulator kernel can be performed.

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:

```
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);
  ...
};
```

In the above code there are three categories of methods: start-stop, simulation control and initial conditions:

- 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.
- Simulation control services allow the controlling of the behavior of the simulation. 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 like queries and modifications. The architectural design of the simulator kernel influences this part of the communications. 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. 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:

```
// 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, out double max,
    out double min, out string des, out string units, out short dim1,
    out short dim2, out short dim3, out string errMsg );
  ...
};
```

In the above code there are several IDL structs with definitions for the simulator data similar to the ones 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 that allowed the incorporation of new types without having to modify the application code.

SimCorba and Receiver have to deal with a huge number of variables (about 26,000), which must be updated at a range of 4 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 ones (using CORBA sequences). The first alternative has advantages such as an easier C++ mapping for programmers, or avoiding problems like lost changes. On the other hand, the huge data volume makes it advisable to use the second alternative (only changes). 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 (changed variables or complete arrays) are available and an application can use either of them. So, for example, if a client application like PPC (see section 6) 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 responsability 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. For this, it uses the operation `writeValues()`. In this operation, the client provides a list of user actions which consist of name-value pairs (struct inputVble) as argument. User actions are performed updating the values of the variables in the simulator kernel and affecting 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:

```
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 like 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, additionally, the clients will have information about which variables are in each position of the arrays (actually, a mapping between each variable and its position). The clients do not need to know anything about SimCorba or the simulator kernel. 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.

```
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

Some of the applications have soft real-time constraints: the refreshing of the screens, the response time of user actions and so on. In order to guarantee predictable behaviour in these cases, we have used the real-time features of RT-CORBA.

In SimCorba, there are thread pools for client invocations. With these thread pools, we can guarantee processor resources for these invocations. Additionally, 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 7).

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 multithread applications. In this sense, there are mutual exclusion zones in different parts of Simcorba, which have been protected through

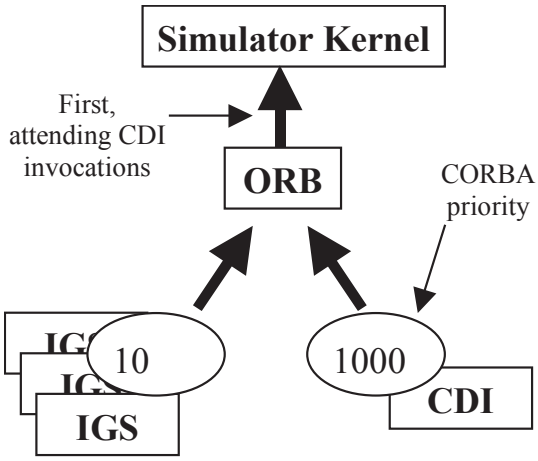

Figure 7. Client applications with different priorities

RT-CORBA mutexes. We can obtain mutual exclusion in these critical parts of Simcorba and, in addition, RT-CORBA mutexes respect the priority of the invocations avoiding priority inversion problems. So, when an application invokes a method that wants 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 cannot simultaneously execute contradictory commands, such as start or stop the simulation.

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. CLIENT TOOLS AND APPLICATIONS

This section presents a brief description of some of the most significant client applications in the simulator context. Communication mechanisms, implemented by the RT-CORBA based SimCorba and Receiver components, are the key elements that facilitate the integration of new tools, offering applications an easy to use programming interface through which they can interact with the simulator kernel.

Variable Debugger DESI: The Variable Debugger DESI 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 future operators. Figure 8 shows a fragment of the DESI tool displaying some variables during a simulation session: `timet` (simulation time), `nstep` (simulation step), `pn` and `fa` (component pressure and area respectively), etc.

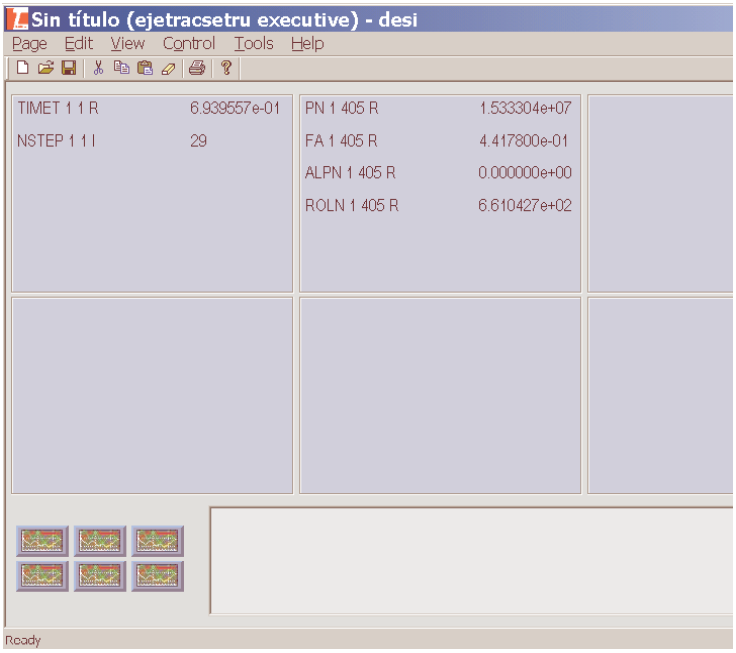

Figure 8. Screenshot of DESI application

Javi: The Javi application (see figure 9) is a 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, they can detect problems easily. The main novelty of this application is the use of Java to provide platform independence. Thanks to CORBA middleware, communications for this Java application are performed in the same way as other C++ based tools.

Supervisor SPV: The Supervisor SPV is another important tool that allows the selection of the simulation models that the kernel is to be composed of, generating both resource and configuration files easily. Some general simulation aspects like cycling time can be adjusted using this tool as well.

IGS Displayers: 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 like opening and closing of valves, alarm recognition, etc. Furthermore, taken actions will have repercussions in the global state of the simulation session, changing conditions that will affect other students.

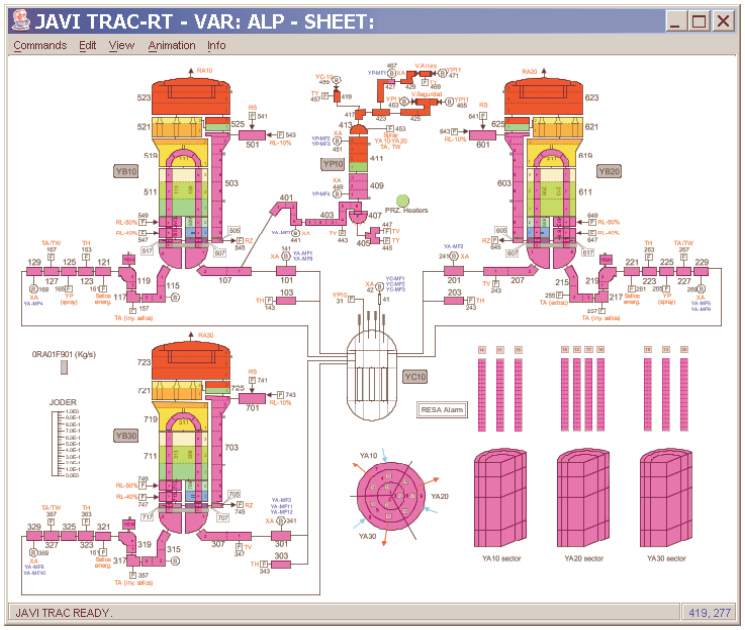

Figure 9. Screenshot of Javi application

IGS Displayers are organized into an instructor-student scheme, where the instructor can perform the same actions as students can, plus additional ones related to the management of the simulation session: control of connected students, modification of special variables, etc. Every IGS post uses a Receiver component to communicate with the simulator kernel. In this context, the Receiver component has to provide data to the IGS Displayers at a rate of 250 milliseconds. Within this rate, the user has the sensation of a correct animation. However, if higher delays occur, students could make errors owing to an incorrect or slow graphical sheet animation.

Plant Process Computer PPC: The Plant Process Computer PPC is an example of a subsystem developed for a specific simulator. The main goal of PPC is the simulation of the Plant Process Computer sited in the power plant. The real 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. Some of the real components have been replicated into the IGS Displayers. So, for example, the keyboards of the PPC have their counterpart in the software keyboards as shown in figure 10.

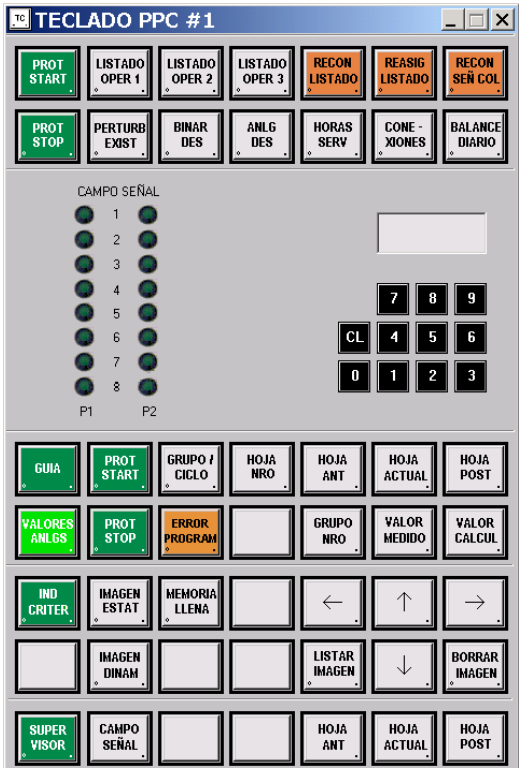

Figure 10. Detail of PPC software keyboard

7. 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 making improvements in 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 leads us to choose a suitable technology for either one.

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 the kernel should be based on had to provide parallel execution and efficient interactions among software components. The high performance computing-oriented Common Component Architecture (CCA) offers these features. The so-called Single Component Multiple Data (SCMD) execution model, provided by the CCA-compliant 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 (the so-called Setru) which simulation models are connected to, allows the construction of different kernels, adapted to different scenarios, turning reusability of scientific code into a reality. 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 different programming teams.

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 Recevier 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 mechanism 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 lots of 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 the most important of these advantages will be strongly manifested in future stages of the simulator software life cycle. Our nuclear power plant simulator is prepared to evolve and to be adapted to different scenarios. 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.

REFERENCES

1. Allan BA, Armstrong RC, Wolfe AP, Ray J, Bernholdt DE, Kohl JA. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience* 2002; **14**(5): 323–345.
2. Alvarez JM, Díaz M, Llopis L, Rus F, Soler E. Practical parallelization strategies of a thermohydraulic code. *Proceedings of Euroconference in Supercomputation in Non Linear and Disordered Systems*, Madrid, Spain 1996; 254–258.
3. Burns A, Wellings A. *Real-Time Systems and Programming Languages* (3th edn). Addison-Wesley, 2001.
4. Components@LLNL: Babel home page. http://www.llnl.gov/CASC/components/babel.html [April 2005].
5. Díaz M, Garrido D. Applying RT-CORBA in nuclear power plant simulators. *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Computer Society: Vienna, Austria 2004; 7–14.
6. Díaz M, Garrido D. A simulation environment for nuclear power plants. *8th IEEE International Workshop on Distributed Simulation and Real-Time Applications*. IEEE Computer Society: Budapest, Hungary 2004; 98–105.
7. Díaz M, Garrido D, Romero S, Rubio B, Soler E, Troya JM. A CCA-compliant nuclear power plant simulator kernel. *Component-Based Software Engineering (CBSE)* (*Lecture Notes in Computer Science*, vol. 3489). Springer: Heidelberg, 2005; 283–297.
8. Díaz M, Rubio B, Soler E, Troya JM. SBASCO: skeleton-based scientific components. *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society: A Coruña, Spain 2004; 318–324.
9. Englander R. *Developing Java Beans*. O'Reilly&Associates, 1997.
10. Geist A, Beguelin A, Dongarra J, Jiang W, Mancheck R, Sunderam VS. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
11. Heineman GT, Councill WT. *Component-Based Software Engineering: Putting the Pieces Together*. Addision Wesley, 2001.
12. Henning M, Vinoski S. *Advanced CORBA Programming with C++*. Addison-Wesley Longman, 1999.
13. Horsmann M, Kirtland M. DCOM Architecture. *Microsoft White Paper*. 1997. Available from http://www.microsoft.com/com/wpaper [April 2005].
14. Lee M, Lee S, Kim KH. Implementation of a TMO-structured real-time airplane-landing simulator on a distributed computing environment. *Software: Practice and Experience* 2004; **34** (15): 1441–1462.
15. Levine DL, Mungee S, Schmidt DC. The design of the TAO real-time object request broker. *Computer Communications* 1998; **21**: 294–324.
16. Monson-Haefel R. *Enterprise Java Beans* (3th edn). O'Reilly&Associates, 2001.
17. Nieplocha J, Harrison RJ, Littlefield RJ. Global arrays: a portable shared memory programming model for distributed memory computers. *Supercomputing'94*. Los Alamitos, CA, USA 1994; 340–349.
18. Object Management Group home page. http://www.omg.org [April 2005].
19. Object Management Group, CORBA home page. http://www.corba.org [April 2005].
20. Object Management Group, specification of CORBA Component Model (CCM) home page. http://www.omg.org/technology/documents/formal/components.htm [April 2005].
21. Schmidt DC, Kuhns F. An overview of the real-time CORBA specification. *IEEE Computer special issue on Object-Oriented Real-time Distributed Computing* 2000; **33** (6): 56–63.
22. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. *MPI: The Complete Reference, volume 1–The MPI Core*. MIT Press, 1998.
23. Tecnatom S.A. home page. http://www.tecnatom.es [April 2005].
24. The Common Component Architecture Forum home page http://www.cca-forum.org [April 2005].
25. Vanneschi M. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing* 2002; **28** (12): 1709–1732.
26. Vivanco RA, Pizzi NJ. Scientific computing with Java and C++: a case study using functional magnetic resonance neuroimages. *Software: Practice and Experience* 2005; **35** (3): 237–254.
27. Zhao H, Georganas ND. HLA real-time extension. *Concurrency and Computation: Practice and Experience* 2004; **16** (15): 1503–1525.