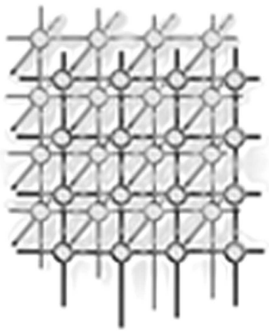
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

A component-based nuclear power plant simulator kernel



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 presents a nuclear power plant simulator kernel based on the high performance computing-oriented Common Component Architecture (CCA). The approach takes advantage of both the component-based software development and the efficient execution of parallel simulation models. The use of components improves the software life cycle and facilitates the development, maintenance and evolution of the simulator kernel, which can be easily adapted to different execution scenarios. Data dependencies among simulation models are automatically resolved by means of a novel algorithm, releasing the programmer from this tedious task and making, as a result, the development process easier. This work introduces the main features of the simulator kernel, describing concepts and the model it is based on. Some preliminary results are shown, which anticipate the feasibility and suitability of the proposal.

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

1. INTRODUCTION

The evolution and growing complexity of current software systems create the need for new programming paradigms that facilitate the development and maintenance of large applications. Component-Based Software Engineering (CBSE) is a modern methodology that proposes the construction of applications by plugging in standalone software components [9]. Based on component interoperability, this programming style allows the creation of more flexible and adaptable software, promoting reusability of components already developed and verified in other projects, which increases the reliability of the final product.

*Correspondence to: Manuel Díaz, Department of Languages and Computer Science, E.T.S. Ingeniería Informática, University of Málaga, 29071 SPAIN
†E-mail: mdr@lcc.uma.es

Initially applied to the business world, component technologies are coming to other areas such as scientific computing. Scientific software frequently demands high performance in order to execute complex mathematical models or simulate physical phenomena in acceptable time. However, components standards and implementations, such as OMG CCM [15], Microsoft DCOM [10], Sun Java Beans and Enterprise Java Beans [7] [12], share serious shortcomings for scientific computing, due to the lack of the abstraction needed by parallel and distributed programming and poor performance. They also have trouble encapsulating an existing scientific application (which might itself be a parallel or distributed application) into a software component. Nowadays, 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 C or FORTRAN and, more recently, Java and C++ [22]. In this sense, ASSIST [21] is focused on high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software. SBASCO [6] combines both parallel skeletons and software components, and 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 [20] to define a standard component architecture for high performance computing.

This work is focused on nuclear power plant simulators. A Pressurized Water Reactor (PWR) plant consist 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.

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 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.

In previous work, we have collaborated with the company Tecnatom S.A. [19] in the development and maintenance of different simulators currently used in several power plants located in Spain, Germany and Mexico [4] [5]. These simulators comprise a collection of software tools and on-line applications that are executed on different network nodes, setting their communications through CORBA [14]. More specifically, the development of distributed applications with soft real-time constraints was based on the Real-time CORBA specification (RT-CORBA) [16]. Other interesting approaches for developing distributed real-time simulators are based on the TMO model [11] and HLA [23].

As the main application in the simulator context, the *simulator kernel* is responsible for executing lots of 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 like TRAC (thermo hydraulic model) or NEMO (neutronic model) to simpler ones simulating, for example, the operation of a valve.

Previous versions of the kernel were 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 same application. 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 tasks (such as the modification, substitution or integration of new simulation models) aimed at adapting the kernel to different scenarios were usually hard to perform. Furthermore, both source code version management and reusability of scientific code were also very limited.

We focus on the idea that a component-based simulator kernel can solve many software maintenance related problems appeared in previous versions of this application. In our approach, 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 controlling the simulation.

Apart from componentization, another additional aspect can 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 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.

In this paper, we present a parallel, component-oriented nuclear power plant simulator kernel. The approach overcomes the above-described limitations, making the management of simulation models easier and allowing the integration of parallel and sequential models into the same simulator. We use an efficient algorithm that automatically resolves all inter-model data dependencies, releasing the programmer from this task and facilitating, as a consequence, the development process. Aiming to combine both componentization and parallelization, we have based the kernel on CCA, a component model specifically designed for high performance computing. The runtime support is based on Ccaffeine [1], a CCA-compliant framework focused on parallel computing environments. Due to the size and large number of simulation models that constitute the global system, the human work force necessary to componentize them is large enough to consider the development of a previous prototype, including some test simulation models, in order to evaluate the feasibility and suitability of our proposal.

The rest of the paper is structured as follows. Section 2 provides an overview of the simulator architecture. Some of the main features of CCA and Ccaffeine are presented in Section 3. Section 4 describes the proposed CCA-compliant simulator kernel and the experimental results obtained by using the prototype. The paper finishes with some conclusions.

2. SIMULATOR OVERVIEW

This section offers an overview of the simulation environment to which the kernel belongs. The simulation projects of Tecnatom S.A. usually include two simulators that influence on hardware and software architectures:

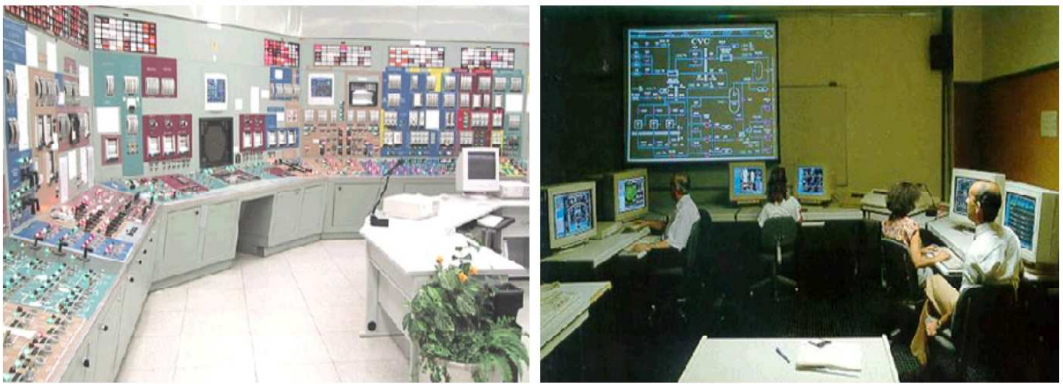


Figure 1. Details of Full Scope Simulator (left) and Interactive Graphic Simulator (right)

- *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.
- 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 group of distributed tools and applications that interact through the high-level communications mechanisms of CORBA. The simulator can be mainly divided into two differentiated parts. The first one, consisted of the simulator kernel and SimCorba, acts as a simulation server offering a set of services to the rest of the client applications, 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:

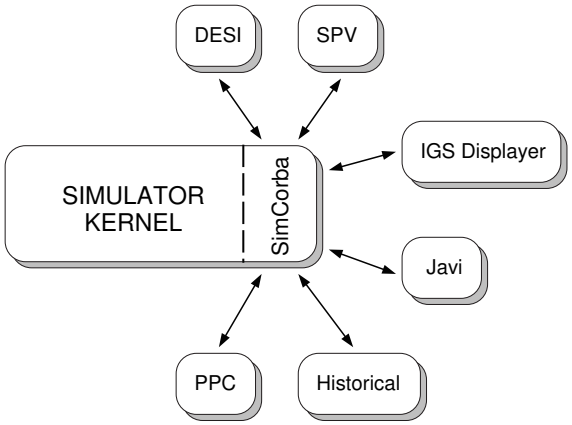


Figure 2. High level software applications conforming the simulator

Simulator Kernel: To compute the simulation of the power plant, the simulator kernel executes simulation models and calculates lots of simulation data, this being the most important application in the simulator. 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 role of SimCorba consists of connecting the 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. CCA FUNDAMENTALS

This section describes some of the basics of the Common Component Architecture (CCA). A more detailed explanation can be found in [20]. 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.

Scientific IDL: The Scientific Interface Definition Language (SIDL) and the Babel tool [3] 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. 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: Frameworks 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. 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 [18], PVM [8] or Global Arrays [13].

4. CCA-COMPLIANT SIMULATOR KERNEL

This section presents the new parallel, component-oriented simulator kernel, describing concepts, architecture, execution phases and performance evaluation tests for the implemented prototype.

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

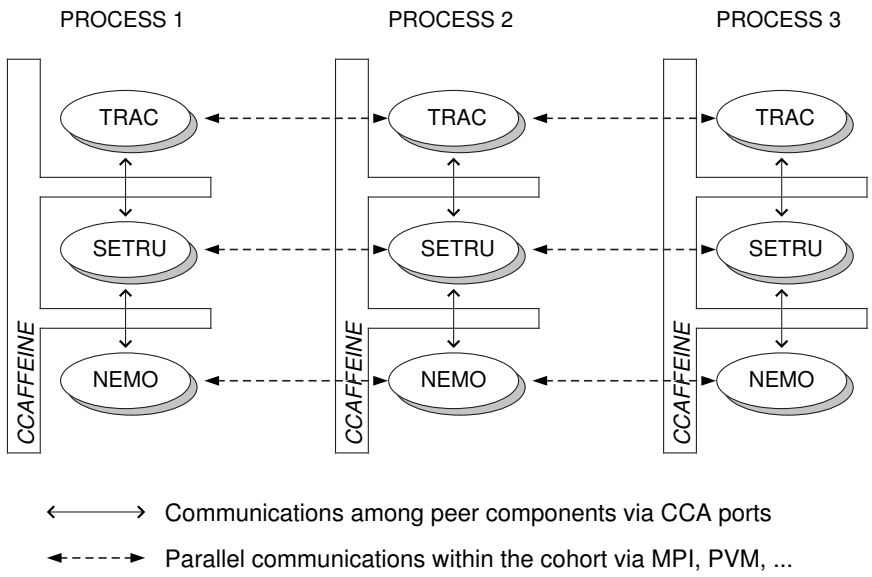


Figure 3. Single Component Multiple Data parallel simulator kernel

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. 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.

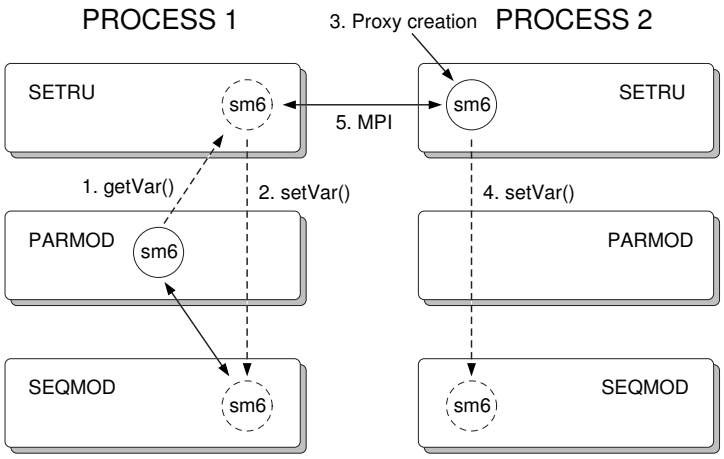


Figure 4. Resolving local and remote data dependencies

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 4 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 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 4 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. 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 which are hosted in other processes and computed by other simulation models.
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 in order to minimize message passing.

4.4. Performance evaluation

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. The purpose of these experiments is to evaluate the efficiency of the mechanisms the described simulator kernel is based on. The developed prototype comprises implementations for Setru component and some test simulation models programmed in C++ with MPI. Our test models implement simple parallel methods for solving partial differential equations (PDEs) based on domain decomposition techniques [17]. We adjust these numeric methods to demand similar amounts of CPU utilization and communications than the real simulation models. The experiments involve comparing the componentized simulator kernel versus a specific implementation made up of a single (non component-oriented) efficiently coded C++/MPI program. We have programmed two different scenarios. The first one (simulation test A) comprises computationally intensive parallel models. Data dependencies among models always occur into the same process (local data dependencies) so message passing is only due to parallel communications within the cohort. The second one (simulation test B) represents a communication intensive scenario in which we reduce CPU demands and add lots of inter-model data dependencies involving different processes (remote data dependencies) in order to make extensive use of the Setru communication mechanisms.

Executions, which are summarized in table I, have been carried out in a cluster of Pentium 4, 2.66GHz, 1GB RAM Linux workstations interconnected with a 1Gb/s Myrinet network. The results show that the penalty overhead (in execution time) imposed by the CCA based implementation are 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 component-oriented paradigm together with the developed runtime system.

5. 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 software evolution, reusability or maintenance, valuable aspects for a simulator that suffers from changes constantly.

The first motivation for using a component-based technology was to overcome these shortcomings making improvements in the simulator kernel software life cycle. Since the

Table I. Execution time (in seconds) and overhead percentage (in brackets) for simulation tests implemented by "classical" C++/MPI program and componentized CCA simulator kernel prototype.

	Sequential	2-Processors	4-Processors
Simulation Test A			
Classical	72.34	40.24	26.30
CCA	75.20	41.92	26.65
	(+3.95%)	(+4.17%)	(+1.33%)
Simulation Test B			
Classical	36.62	26.95	20.15
CCA	37.43	28.16	20.82
	(+2.21%)	(+4.48%)	(+3.32%)

majority of computationally intensive simulation models of our system can be parallelized to reduce their execution time, and due to the fact that performance is a major requirement for this application, the chosen component model should provide parallel execution and efficient interactions among software components. The high performance computing-oriented Common Component Architecture (CCA) offers these features. 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.

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. 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. Finally, we have implemented a prototype and we have presented some experimental results that anticipate the feasibility of the proposal.

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. Components@LLNL: Babel home page. http://www.llnl.gov/CASC/components/babel.html [April 2005].
4. 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.
5. 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.
6. 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.
7. Englander R. *Developing Java Beans*. O'Reilly&Associates, 1997.
8. Geist A, Beguelin A, Dongarra J, Jiang W, Mancheck R, Sunderam VS. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
9. Heineman GT, Councill WT. *Component-Based Software Engineering: Putting the Pieces Together*. Addision Wesley, 2001.
10. Horsmann M, Kirtland M. DCOM Architecture. *Microsoft White Paper*. 1997. Available from http://www.microsoft.com/com/wpaper [April 2005].
11. 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.
12. Monson-Haefel R. *Enterprise Java Beans* (3th edn). O'Reilly&Associates, 2001.
13. 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.
14. Object Management Group, CORBA home page. http://www.corba.org [April 2005].
15. Object Management Group, specification of CORBA Component Model (CCM) home page. http://www.omg.org/technology/documents/formal/components.htm [April 2005].
16. 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.
17. Smith B, Bjorstad P, Gropp W. *Domain Decomposition. Parallel Multilevel Methods for Elliptic P.D.E.'s*. Cambridge University Press, 1996.
18. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. *MPI: The Complete Reference, volume 1–The MPI Core*. MIT Press, 1998.
19. Tecnatom S.A. home page. http://www.tecnatom.es [April 2005].
20. The Common Component Architecture Forum home page http://www.cca-forum.org [April 2005].
21. Vanneschi M. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing* 2002; **28** (12): 1709–1732.
22. 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.
23. Zhao H, Georganas ND. HLA real-time extension. *Concurrency and Computation: Practice and Experience* 2004; **16** (15): 1503–1525.