A CCA-compliant Nuclear Power Plant Simulator Kernel

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

Dpto. Lenguajes y Ciencias de la Computación. Málaga University
29071 Málaga, SPAIN
{mdr, dgarrido, sromero, tolo, esc, troya}@lcc.uma.es

Abstract. This paper presents a parallel, component-oriented nuclear power plant simulator kernel. It is based on the high-performance computing oriented Common Component Architecture. The approach takes advantage of both the component paradigm and the parallel execution of simulation models. This way, the maintenance, evolution and efficiency of a simulator are improved. The work introduces the main features of the simulator kernel, describing concepts and the model it is based on. Data dependencies among components (simulation models conforming a simulator) are solved in a configuration phase, reducing the execution time of the simulation phase. Some preliminary results are shown, which anticipate the feasibility, suitability and efficiency of the proposal.

1 Introduction

The evolution and growing complexity of modern software systems create the need for new programming paradigms that facilitate the development and maintenance of software applications. Component-Based Software Engineering proposes the development of applications by plugging 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 and increasing, this way, the reliability of the final product.

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. Components standards and implementations, such as OMG CCM [14], Microsoft DCOM [10], Sun Java Beans and Enterprise Java Beans [7] [11], share serious shortcomings for parallel and distributed scientific applications, due to the lack of the abstraction needed by parallel and distributed programming and poor performance. They also have trouble with the mechanism for encapsulating an existing scientific application (which might itself be a parallel-distributed application) into a component.

Recently, some efforts are being carried out in order to incorporate component technologies into high performance computing area. In this sense, ASSIST [19] is focused on high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software. SBASCO [6] is oriented to the efficient development of parallel and distributed numerical applications. A large effort is currently devoted by the Common Component Architecture (CCA) forum [18] 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. In order to simulate the operation of a nuclear power plant in a computer system, we need detailed models of heat transmission, vessel, valves, pipes and pumps, etc. Many simulation codes use a plant nodalization as input, a model built up by interconnecting a set of predefined cells whose state variables are solved every time step.

The use of simulators has special importance in the context of nuclear power plants. On the one hand, they can predict the plant status when facing to different situations that can occur in the daily operation. In this sense, a fast response is required and performance becomes a major factor. On the other hand, they can be used as training tools for future operators, allowing for the practice of normal (temperature monitoring, valve manipulation) and unusual (emergency) situations.

Currently, we are collaborating with Tecnatom S.A. [17] in order to carry out the maintenance of different nuclear power plant simulators [4] [5]. Software architecture of these simulators is organized as a collection of distributed applications that can be executed on any node of a network, setting their communications through CORBA [13]. The *simulator kernel* is the most important application in the simulator context. It carries out the iterative execution of different *simulation models*, which are responsible for the precise simulation of physical components of the real system. There is a wide range of simulation models, from computationally intensive complex models like TRAC (thermo hydraulic model) or NEMO (neutronic model) to simpler ones simulating, for example, the operating of a valve.

Current simulators are programmed in a classical (non component-oriented) style, in such a way that simulation models, coded as sequential Fortran subroutines, are statically linked together into the simulator kernel. This approach, apparently feasible at first, presents 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 others. In order to resolve data dependencies among models, all shared data are declared as global variables allowing access from any subroutine. Besides that, it is difficult to manage different versions of

the same models, and substitution, modification or integration of new models into the simulator turn into tedious tasks. Reusability of these simulation models in other simulators is also limited due to the used programming techniques and the coupling among the current models.

An attractive solution for solving all these software maintenance related problems is componentization. In this new approach, a simulator kernel can be constructed by connecting the corresponding simulation models, now encapsuled into software components, to a central manager component which implements the runtime system taking care of executing the models in a proper way. Apart from componentization, an additional aspect can be taken into account in order to improve the system. In the current simulators, simulation models are implemented using sequential procedures. However, codes from some of these models can be parallelized by splitting the computation in such a way that they could be run on multiple processors in order to reduce their execution time. For example, in [2] a parallel version of the thermo hydraulic code encapsulated into the TRAC simulation model is described. Parallelization 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 version of the simulator kernel of a nuclear power plant simulator. 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. This way, the maintenance, evolution and efficiency of a simulator are improved. We have chosen CCA, a component model specifically designed for high performance scientific computing, as component technology for programming the simulator kernel. 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 runtime support is based on Ccaffeine [1], a CCA-compliant framework oriented to high performance parallel computing environments.

The rest of the paper is structured as follows. Section 2 introduces the software architecture of the simulators. The main features of the CCA component model 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 and future work.

2 Simulator Architecture

The simulation projects of Tecnatom S.A. usually include two simulators that influence on hardware and software architectures. The first one is called Interactive Graphic Simulator (IGS), which through graphic applications (see figure 1, right) allows operator training. The second one is called Full Scope Simulator (FSS), which is an exact replica of the power plant control room taking care

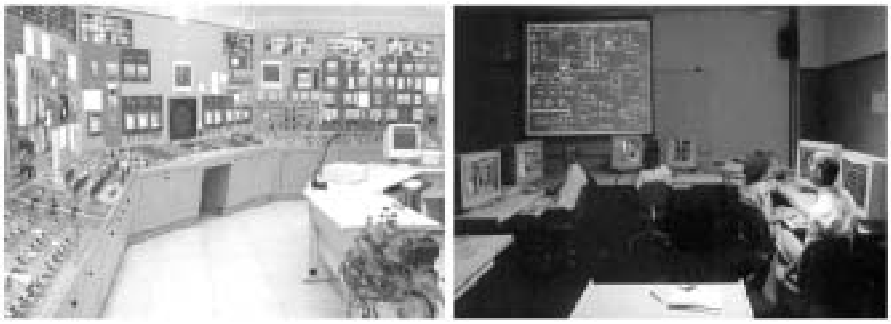

Fig. 1. Details of *Full Scope Simulator* (left) and *Interactive Graphic Simulator* (right).

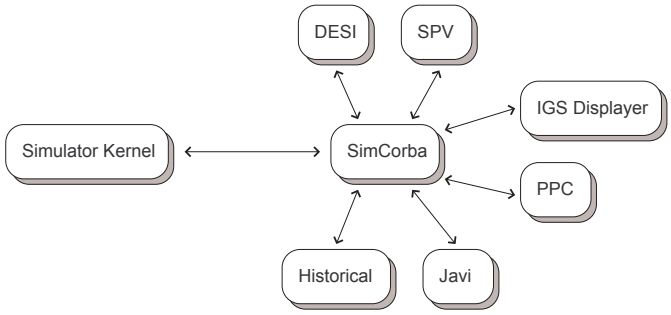

Fig. 2. Main software applications conforming the simulator.

of all details, from physical artifacts such as furniture, control panels, etc. to software simulating the applications running in the room (see figure 1, left).

The main high level hardware elements of FSS and IGS are: Simulation computers, Instructor console, Physical panels and Student workstations. Simulation computers are responsible for the simulation process executing the simulation models and providing data to the rest of software and hardware components. 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. The Physical panels are exact replicas of those existing in the control room. Operators carry out their actions mainly through these panels, with hundreds of indicators, hardware keyboards, etc. IGS simulators additionally include the hardware needed for 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 software architecture of the entire simulator is organized as a collection of distributed applications that interact with each other through the high-level communication mechanisms provided by CORBA. Some of the most important applications together with their interactions can be seen in figure 2. The complete

software system can be divided into two well differentiated parts. The first one comprises the simulator kernel and SimCorba. These two applications act as a simulation server offering a set of simulation services to the rest of tools and applications, which constitute the second part of the system. The following is a short description of the applications conforming the simulator:

- **Simulator Kernel:** application responsible for computing the simulation of the nuclear power plant by executing the different simulation models. It is the most important application in the simulator context.
- **SimCorba:** a simulation server that offers a set of services such as periodic transfer of variables, actions over the simulator, etc. to the rest of applications. It manages all communications between client applications and the simulator kernel.
- **Client applications:** a wide group of applications that interact with SimCorba for different purposes, such as debugging the simulation process, allowing representation and modification of simulation variables, changing simulation aspects like cycling time, recording the simulation state in real-time including all significant variables, etc.

Due to the system size and the heterogeneity of the involved applications, the development of simulators includes different platforms such as Unix, Linux and Windows, with different programming languages such as C++, Java and Fortran.

The work presented in this paper is focussed on one concrete application from the above-described architecture: the simulator kernel. The shortcomings related to the current simulator kernel design, as mentioned before, encouraged us to develop a new version of this application based on component-oriented programming and parallel execution of simulation models. The new simulator kernel, together with SimCorba, must implement the same set of simulation services provided by the current version in order to keep the rest of client applications unchanged.

3 CCA Model

This section describes the main features of the Common Component Architecture (CCA). A more detailed explanation can be found in [18]. CCA provides a means for scientific software developers to build applications by assembling software components in a "plug and play" environment for high performance computing. CCA is a specification developed by the CCA Forum to describe the rules for constructing CCA components, the model for linking them together and the collection of services that frameworks should provide.

Components interact with each other through well-defined *ports* which are the key elements of the connection model representing communication end point for components. A CCA port is described by an interface which declares a collection of methods without revealing implementation details. In this sense, they are similar to interfaces in Java or abstract virtual classes in C++. Components are

linked together by connecting their ports following a *provides-uses* interface design pattern similar to that within the OMG CCM proposal. According to this, 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 functionality a component may need and are the stubs that a component uses to invoke services provided by another component. A uses port of a component can be attached to a compatible (same type) provides port of another component. Since this connection is performed, a procedural (not dataflow) relationship is established and any functionality represented by the uses port is obtained by invoking the methods in the connected provides port.

The Scientific Interface Definition Language (SIDL) and the Babel tool [3] have been adopted by CCA in order to make the use of components independent of their implementation languages. SIDL is the 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. It also provides other useful features such as enumerated types, symbol versioning, name space management as well as an object model with 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, making it possible to connect together into the same application components implemented in different languages.

A framework represents a concrete implementation of the model mechanisms. Component instances are created and managed within a framework, which must provide, according to the CCA specification, a minimal set of services that components use in order to communicate with each other. Currently, different frameworks have been developed to support specific computational environments such as parallel, distributed or multithread. Ccaffeine, the CCA-compliant framework employed in this work, 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 on 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 for calling 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 [16], PVM [8], Global Arrays [12] or Shared Memory. CCA and Ccaffeine make it possible to construct high performance applications by

connecting parallel components which are (possibly) implemented in different programming languages and even making use of distinct parallel communication libraries.

4 CCA-compliant Simulator Kernel

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

4.1 Concepts and SIDL definitions

Simulation models contain the necessary code to simulate specific parts of the system. The execution of a simulation model usually requires reading or updating data variables which are computed by other models (we also refer to these data as *simulation variables*). The previous (classical) version of the simulator kernel resolved these types of inter-model data dependencies by declaring all shared data as global variables allowing access from any subroutine. 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 component must report on:

- *Simulation variables it needs*, for reading or updating, from other models in order to be executed.
- *Simulation variables it provides*, which are computed and offered (exported) to the rest of models.

According to this, the programmer of a concrete simulation model component only has to declare, by implementing the corresponding methods, the simulation variables needed from/provided to the rest of models whereas a *manager component* integrated in the kernel is the one responsible for locating the requested variables (even if they are hosted in different processes) and providing them to the respective models. Specific inter-process communication schemes needed to resolve data dependencies efficiently are established in a configuration phase. This way of managing data dependencies leads to a significant uncoupling among simulation models in both development and execution time. The programmer is not concerned about issues such as knowing the rest of models in the simulator or resolving data dependencies and so, he/she can be focussed on writing scientific code for the simulation model under development.

Simulation models are now encapsulated into (sequential or parallel) independent software components. Since all of them are CCA-compliant components implementing a concrete interface, we make sure they can be integrated into the same simulator, even if they are programmed using different languages or communication libraries. Our proposal makes the construction of different simulator

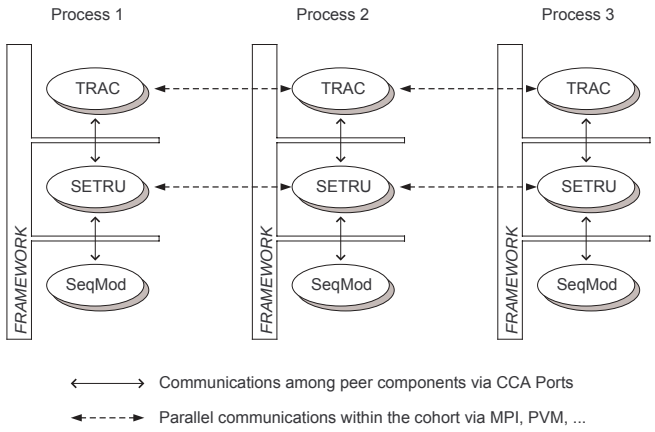

Fig. 3. Simulator Kernel implemented as a SCMD-parallel application.

kernels possible. This can be easily achieved by selecting and composing the corresponding simulation models from a component repository. All simulator kernels developed in this way share the same architecture that comprises a central manager component called SETRU and the appropriate collection of simulation models connected to it. SETRU takes care of controlling the simulation by executing the different simulation models in a proper way. In fact, this component plays a major role since it implements the entire runtime system of the simulator kernel. Some of the most important functions carried out by SETRU are:

- To retrieve information from the models connected to it, setting up data structures accessed during the simulation.
- To resolve data dependencies, providing the requested simulation variables to the corresponding models.
- To execute the different simulation models properly.
- To send values of updated simulation variables to the rest of processes for maintaining data consistency.
- To create additional threads for handling communications with SimCorba and executing the simulation commands received from it.

Figure 3 describes the Single Component Multiple Data paradigm used to implement the simulator kernel as a Ccaffeine application. In this paradigm, all components together with the framework are instantiated in every participant process. Different components in the same process communicate among them through the CCA Port mechanism, which occurs, for example, when SETRU needs to call methods on models connected to it.

Simulation models can be programmed using both sequential and parallel programming styles. Instances in different processes of the same parallel component, e.g. TRAC in figure 3, communicate with each other by using a parallel

```
package simkernel version 1.0
{
  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);
    void setValue(in int cell, in double value);
    array<double> getAllValues();
    void setAllValues(in array<double> values);

    SimVariable subVar(in int initCell, in int finalCell);
    void assign(in SimVariable variable);
    string toString();
  }

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

    void setup();
    void initialize();
    void execute();
  }

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

Fig. 4. SIDL definitions for SimVariable class and ISimModel component interface.

communication library such as MPI or PVM. In this case, all simulation variables computed by the model are distributed across the different processes. On the other hand, instances of a component that represents a sequential simulation model, e.g. SeqMod in figure 3, do not communicate among them and the same whole computation is executed on every process, having their simulation variables replicated on all processes. Other situations can also be considered: for example, a parallel simulation model having both distributed and replicated data variables, or a sequential model being executed on one process only (not replicated). The proposed simulator kernel supports the integration of all these different types of simulation models together, maintaining data consistency for both distributed and replicated data.

Simulation codes model the nuclear power plant as a set of interconnected *nodes* and *cells*. We represent simulation variables, which are used to refer to these nodes and cells, as instances of SIDL classes (figure 4). An instance of class SimReference has a variable name, a number of node and a range of cells over which the particular variable is computed, for example, *pressure calculated on node 2, cells from 1 to 5*. SimVariable class extends SimReference to add the specific real value that the variable takes on each cell. Simulation models use lists of SimReference objects to declare simulation variables needed from other models. The values of these requested variables are provided by SETRU by means of SimVariable objects.

All simulation models are encapsulated into CCA components that implement the ISimModel interface shown in figure 4 and described in next section. Since some methods of ISimModel are independent of the specific simulation model, we offer a base class, called BaseModel, that implements them. Taking advantage of the mechanisms provided by SIDL, classes implementing simulation model components can (optionally) inherit from this base class. In this case, the programmer only needs to code specific methods for configuring, initializing and executing the specific simulation model.

4.2 Execution phases

The simulator kernel execution is divided into two different phases. In the first one, called the *configuration phase*, both simulation models and communications with SimCorba are properly configured. In the second one, called the *simulation phase*, the execution of the models is carried out according to the simulation commands received from SimCorba.

Configuration phase. The structure of a concrete simulator kernel, including the simulation models employed, their relative execution order and a set of simulation parameters, is described in a configuration file. SETRU component reads this file and registers a ISimModel *uses port* for each simulation model composing the simulator in order to communicate with them. On the other hand, simulation models only needs to register one ISimModel *provides port* (to provide services to SETRU), besides the proper *uses ports* needed to use functionality offered by other auxiliary components. Port registration procedure is carried out by the *setService()* method, which must be implemented by every component according to the CCA specification. This method is called by the framework when the component is instantiated.

Once all components are connected together, SETRU calls *setup()* and *initialize()* methods on each simulation model. The former contains the necessary code to create and configure the simulation variables provided (computed) by the model as well as the lists of SimReference objects representing the sets of variables read, updated and provided. The latter initializes the provided variables with correct values. Then, SETRU calls *getListRefRead()*, *getListRefUpdated()* and *getListRefProvided()* methods on the connected models. These methods return the lists of SimReference objects created by *setup()*.

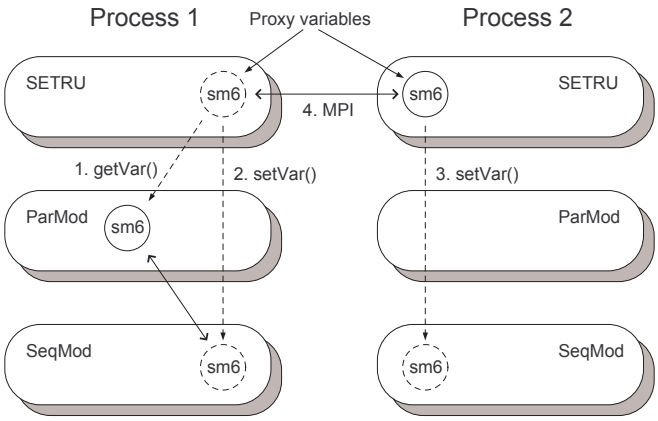

Fig. 5. All instances of *SeqMod* read the simulation variable *sm6* provided by *ParMod* in process 1. *SETRU* uses MPI for updating proxy variables in all processes.

Since information obtained locally from the connected models is sent to all participant processes, SETRU component in each process knows the location (process and model) of simulation variables, which lets it to resolve local and remote data dependencies:

- **Local data dependencies.** When a simulation model needs a variable computed by other model in the same process, SETRU provides it by calling *getVar()* on the "provider" model and *setVar()* on the "requester". An example can be seen in figure 5, where SeqMod in process 1 needs to read the simulation variable *sm6* provided by ParMod in the same process. By calling *getVar()* on ParMod, SETRU obtains a SimVariable object representing the simulation variable, which is passed to SeqMod through *setVar()* method. Since SIDL objects are implemented as references (pointers) by the respective programming languages, SimVariable instances used in both SeqMod and ParMod models make reference to the same physical data.
- **Remote data dependencies.** However, if data dependencies involve different processes, an additional intermediate (proxy) variable is created in every process. When the simulation variable is modified due to the execution of a particular simulation model, its new value is sent to the corresponding processes by using these proxy variables. Figure 5 shows this type of data dependencies as well. SeqMod instance in process 2 is the one that reads variable *sm6* exported by ParMod in process 1. SETRU creates proxy variables on both processes. The SimVariable object representing *sm6* is obtained by calling *getVar()* on ParMod and assigned to the proxy variable in process 1. On the other hand, the proxy created in process 2 is passed to SeqMod instance in the same process by calling *setVar()* method. During the simulation, SETRU uses message passing for updating proxy variables in different

processes with the values computed by the provider model ParMod. This way, simulation models can treat proxies as if they were local simulation variables, reading or updating its values, while SETRU takes care of hiding all communication details.

Since the main execution thread carries out the simulation, additional threads are needed in order to communicate with SimCorba application. At the end of the configuration phase, SETRU creates these threads and initialize CORBA communication mechanisms.

Simulation phase. Each time a simulation model is going to be executed, its required simulation variables must be updated with the latest computed values. By using information obtained in the configuration phase, a specific inter-process communication pattern can be established for each particular model in order to determine the minimal MPI communications needed to resolve its remote data dependencies. By using these fixed communication patterns, message passing is carried out efficiently during the rest of the simulation.

From this point, the simulator kernel is ready to react to different simulation commands received from SimCorba such as *start/stop simulation*, *start/stop debugging mode*, *execute n steps*, *query/modify simulation variable*, etc. Once the proper command has been received, the simulation begins with the execution of the models in the relative order described in the configuration file. The execution of a simulation model involves the following three steps that are carried out by SETRU:

- **Step 1:** Proxy variables are updated with the values of simulation variables hosted in other processes and needed by the model. Values of all simulation variables sent from one process to another are previously packed together in order to minimize message passing.
- **Step 2:** The simulation model is executed by calling its *execute()* method in every process. This method contains the parallel or sequential code implementing the simulation.
- **Step 3:** Once the model is executed, updated values corresponding to simulation variables hosted in other processes are sent to them in order to modify variables in the provider models.

4.3 Performance evaluation

The use of CCA components together with the Ccaffeine framework, SIDL/Babel tool for language interoperability and (specially) the defined runtime system, which allows the integration and communication of generic simulation models, may affect the efficiency of the system. 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. We consider

Table 1. Execution time (in seconds) and overhead percentage (in brackets) for simulation tests implemented by "classical" C++/MPI program and componentized CCA simulator kernel prototype.

Simulation Test A (Classical vs. CCA)			
	Sequential	2 Processors	4 Processors
Classical	72.34	40.24	26.30
CCA	75.20 (+3.95%)	41.92 (+4.17%)	26.65 (+1.33%)

Simulation Test B (Classical vs. CCA)			
	Sequential	2 Processors	4 Processors
Classical	36.62	26.95	20.15
CCA	37.43 (+2.21%)	28.16 (+4.48%)	20.82 (+3.32%)

appropriate for our test models to implement simple parallel methods for solving partial differential equations (PDEs) based on domain decomposition techniques [15]. We can adjust these numeric methods to demand the same CPU utilization and communications than the real simulation models. The experiments involve comparing the componentized simulator versus a specific direct 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 SETRU communication mechanisms.

Executions have been carried out in a cluster of Pentium 4, 2.66GHz, 1GB RAM Linux workstations interconnected with a 1Gb/s Myrinet network. Table 1 summarizes the obtained results, which are very similar for the two implementations in both simulation tests. This behavior is due to:

- Component interactions through CCA ports are performed efficiently in the context of Ccaffeine framework.
- Communication schemes, automatically calculated in the configuration phase, are very similar to those coded in the classical approach in order to resolve inter-model remote data dependencies.
- Models use *getAllValues()* method of SimVariable to obtain an array object containing cell values. SIDL array classes have a method which returns a

direct pointer to stored data. By calling this method for every simulation variable in the configuration phase, efficient data access is carried out during the simulation.
- When reasonable amounts of computation take place into the models, which is usual for the described simulators, the SETRU runtime system overhead can be accepted.

As a first approach, results reveal the feasibility of our proposal, in the sense that the minimal penalty overhead imposed by the CCA implementation (less than 5% in both simulation tests) is compensated for by the advantages of the component-oriented paradigm together with the developed runtime system.

5 Conclusions and Future Work

A nuclear power plant simulator is a complex, computationally intensive application that can take advantage of component-based software engineering, especially when the used component model allows the execution of parallel components in an efficient way. This paper has presented a CCA-compliant simulator kernel, where simulation models are implemented as both sequential and parallel components. The main characteristics of the used model have been introduced. Simulation models export the variables they own and declare the variables they need. The kernel is in charge of solving data dependencies among components in the configuration phase. This way, the programmer is released from managing inter-model communications and the simulation phase is carried out in a more efficient way. Some preliminary results have shown the suitability of the proposal.

A full scope nuclear power plant simulator needs the integration of hundreds of simulation models. Currently, most of these models are developed by using a graphical tool, which is going to be modified to generate the CCA-compliant models in order to be integrated into the described simulator kernel.

References

1. Allan, B.A., Armstrong, R.C., Wolfe, A.P., Ray, J., Bernholdt, D.E., Kohl, J.A., The CCA Core Specification in a Distributed Memory SPMD Framework, *Concurrency and Computation: Practice and Experience*, **14**, 5 (2002), pp. 323–345.
2. Alvarez, J.M., Díaz, M., Llopis, L., Rus, F., Soler, E., Practical Parallelization Strategies of a Thermohydraulic Code, in *Proceedings of Euroconference in Supercomputation in Non Linear and Disordered Systems*, pp. 254–258, Madrid, Spain, 1996.
3. Components@LLNL: Babel, home page http://www.llnl.gov/CASC/components/babel.html.
4. Díaz, M., Garrido, D., Applying RT-CORBA in Nuclear Power Plant Simulators, in *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pp. 7–14, IEEE Computer Society, Vienna, Austria, 2004.

5. Díaz, M., Garrido, D., A Simulation Environment for Nuclear Power Plants, in 8th IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT 2004), pp. 98–105, IEEE Computer Society, Budapest, Hungary, 2004.
6. Díaz, M., Rubio, B., Soler, E., Troya, J.M., SBASCO: Skeleton-Based Scientific Components, in *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2004)*, pp. 318–324, IEEE Computer Society, A Coruña, Spain, 2004.
7. Englander, R., Developing Java Beans. O'Reilly&Associates, 1997.
8. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., Sunderam, V.S., PVM: Parallel Virtual Machine. MIT Press, 1994.
9. Heineman, G.T., Councill, W.T., 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.
11. Monson-Haefel, R., Enterprise Java Beans 3th edition, O'Reilly&Associates, 2001.
12. Nieplocha, J., Harrison, R.J., Littlefield, R.J., Global Arrays: a Portable Shared Memory Programming Model for Distributed Memory Computers, in *Supercomputing'94*, pp. 340–349, Los Alamitos, California, USA, 1994.
13. Object Management Group, CORBA home page http://www.corba.org.
14. Object Management Group (OMG), Specification of Corba Component Model (CCM). http://www.omg.org/technology/documents/formal/components.htm.
15. Smith, B., Bjorstad, P., Gropp, W., Domain Decomposition. Parallel Multilevel Methods for Elliptic P.D.E.'s. Cambridge University Press, 1996.
16. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., MPI: The Complete Reference, volume 1–The MPI Core. MIT Press, 1998.
17. Tecnatom S.A. home page http://www.tecnatom.es
18. The Common Component Architecture Forum, home page http://www.cca-forum.org.
19. Vanneschi, M., The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications, *Parallel Computing*, **28**, 12 (2002), pp. 1709–1732.