Seamless Integration of Heterogeneous Components in a Distributed Simulation Infrastructure

[1]BRAULIO ADRIANO DE MELLO
[2]FLÁVIO RECH WAGNER

[1]Universidade Federal de Lavras – DCC/UFLA – Lavras, MG
[2]Universidade Federal do Rio Grande do Sul – PPGC/UFRGS, Porto Alegre, RS
[1]bmello@dcc.ufla.br [2]flavio@inf.ufrgs.br

The utilization of simulation in the design of systems combining different parts (heterogeneous systems) contributes with the validation, verification, and observation of the behavior of these parts (individually) and of the cooperation between them. The differences between the parts (elements) that compose the model of representation (of a system) make these tasks more complex due to three main aspects: the reuse, the preservation of elements and the interface compatibility. The reuse of existing elements can facilitate and accelerate the construction of new models or the modification of existing ones. Nevertheless, the probability of having a heterogeneous model increases since the elements can be developed under conditions that vary in terms of description language, interface configuration, operational environment, and others. Thus, it is highly likely the need of making an element have alterations or adaptations that can jeopardize the preservation of its original characteristics or even interfere in intellectual properties issues, in order to be reused. The compatibility between interfaces of different elements is neither guaranteed. Regarding these challenges, this paper presents DCB (Distributed Co-simulation Backbone), an infrastructure for distributed simulation that offers mechanisms for preserving elements of the model from interface adaptations and for implementing data translation and communication and synchronization, all of them necessary for the execution of a heterogeneous model. By implementing these aspects in a backbone that is independent from the elements, DCB can preserve their integrity without loosing the capability of adaptation of heterogeneous elements so as to be able to cooperate correctly when integrated into the same simulation model. DCB also permits the execution of a simulation with elements distributed in different nodes.

Keywords: Heterogeneous Models, Distributed Simulation, Integration, Element

(Received October 30, 2008 / Accepted March 16, 2009)

1 Introduction

Simulation allows the emulation and observation of the behavior of real systems by executing representation models. Currently, the growing size and complexity of systems in different domains leads to design and implementation techniques that are based on the reuse of previously available and tested elements (parts of a system). These elements may come from different sources, including third-part ones, and may be designed and implemented without previous knowledge of other elements with which they must cooperate in a given environment. This leads to the utilization of heterogeneous models as an alternative to represent the system behavior with fidelity.

A representation model is called *heterogeneous* when it combines elements that differ in their implementation technology and/or present incompatible functional interfaces. The technology for the implementation or description may correspond to distinct specification languages or techniques [16]. Incompatible interfaces occur when the interaction resources of an element (for instance regarding the form, number, type, and size of these resources, usually available in terms of attributes, protocols, or methods) do not match the resources of a second element with which it must communicate and cooperate.

Heterogeneous simulation is an approach that offers support to the cooperation between heterogeneous elements of a simulation model. As the difficulty in the utilization of given standards for the design and specification of element interfaces and behavior grows, because of the diversity and complexity of these elements, the benefits of heterogeneous simulation also increase.

As an example, in precision agriculture [14], heterogeneous models are useful in the system description and simulation before test fields that evaluate the cooperation between agricultural equipment and the precision systems that are coupled to them. Similar benefits can be identified in wireless sensor network systems (WSNs) [8], where there is a great diversity of cooperating elements, with varying capabilities, purposes, and formats.

The distributed simulation of heterogeneous models is useful in situations where it is not possible or interesting, for different reasons, to have all elements of a model in a single node of a computer network. In some situations, there are access restrictions to given elements, for instance because of intellectual property issues, when an element provider does not want to disclose the element code. In this case, the element must be remotely integrated into the simulation model through its interface. It may also happen that the simulation of a given element requires resources that are available only in a given node of the network. A third situation is the integration of a real-life element to a simulation model, through a special interface that is available in a given node.

In the design and implementation of heterogeneous systems, very often elements are developed and tested independently from each other, usually also using distinct design processes and tools. Because of this situation, the validation of the individual elements cannot consider their interaction with the rest of the system, and the first integration tests are performed only with the real system and in a very late design stage. This delays the detection of problems that could be solved with much smaller costs in early design stages [17], if appropriate tools were available. In this context, heterogeneous distributed simulation techniques offer convenient alternatives for the validation of individual elements and for observing the interactions between them [6].

Although the research in the field of heterogeneous distributed simulation contributes to the consolidation of various fundamental issues, for instance related to communication, synchronization, performance, and modeling, existing solutions in general follow a proprietary approach and are developed for solving problems in a particular domain. As an example, SensorSim [12] is dedicated to the simulation of sensor networks and does not provide mechanisms for ensuring communication with other tools, even targeted to the same domain.

The High Level Architecture (HLA) standard [11] is a remarkable reference in distributed simulation. Although HLA has very interesting characteristics regarding distribution, it does not administrate the interoperability between elements (or *federates* as referred in HLA) that were developed for different RTI (Run-Time Infrastructure) implementations. The RTI is responsible for the cooperation between federates of the same federation. This characteristic led to studies in order to find ways to solve interoperability and reuse issues regarding federates whose interfaces have been implemented for different RTIs [9].

Considering current restrictions and trends of heterogeneous distributed simulation techniques, this paper presents the Tangram/DCB (*Distributed Co-simulation Backbone*) environment. DCB is a backbone for heterogeneous distributed simulation. Tangram is a modeling front-end for DCB, which has been designed to facilitate the specification of heterogeneous models according to the DCB principles.

The DCB architecture has been designed to preserve the original descriptions of the behavior and interface of individual elements. Differently from other approaches, DCB mechanisms remain completely hidden from the elements, which may thus preserve their proprietary solutions and do not need to be modified in order to be integrated into the simulation model. This promotes a much larger reusability of elements than other approaches. In a model of a sensor network, for instance, a new sensor node with a different interface may be added to a model without requiring adaptations in its interface or in the interface of other sensor nodes with which it will communicate.

This property is maintained essentially by the complete encapsulation, within DCB's own modules, of strategies for distributed simulation and for interaction between heterogeneous elements. These strategies consider the required adaptation between different languages and interfaces used in the design of the individual elements, such that the elements remain unaware of them. DCB implements the concept of *gateway* for handling and adapting diverse elements' languages and interfaces. This may be combined with interface *wrappers*, which adapt the interface functionality, when required.

The complete independence between elements is an essential aspect also in the DCB's approach for handling distributed simulation. An element does not need to maintain information on other elements that interact with it. This information is kept inside DCB's own modules. The same approach is used to implement the concept of hybrid synchronization. DCB allows the interaction between elements that follow different synchronization approaches – synchronous, asynchronous, and *no-time* (elements without explicit handling of simulation time), but this feature remains hidden from the elements.

The combined utilization of these characteristics – independence between elements, preservation of reused elements, and hybrid synchronization – offers benefits for heterogeneous distributed simulation that are not found in other environments or tools. This paper, however, will concentrate on the aspects of integration and preservation of heterogeneous elements, involving the required adaptations, encapsulated in the DCB modules, to handle models with diverse languages and interfaces. Heterogeneous simulation is largely explored in the context of electronic embedded systems [13]. A prototype of the Tangram / DCB environment has been developed for this domain, considering its main requirements but maintaining the general principles of the DCB architecture.

The remaining of this paper is organized as follows. Section 2 discusses related work and gives the motivation for the development of DCB. The DCB principles and architecture are presented in Section 3. The Tangram front-end is described in Section 4. Section 5 introduces the DCB prototype targeted at the embedded systems domain. A case study for the validation of this prototype is presented in Section 6. Section 7 draws main conclusions and discusses future work.

2 Related Work

HLA (High-Level Architecture) [5] is an architecture defined by the IEEE 1516 standard, which specifies rules and interfaces for distributed simulation handled by a run-time layer called RTI (Run-Time Infrastructure). Based on the concepts of *federation* and *federates*, this architecture is defined by three main components: a common formal model for specification; a set of services described in the RTI; and a set of functional rules for simulation. A federation is a collection of federates. Federates may be simulators, sub-systems, or other systems. Federations are built according to a formal model that, besides considering the specification of the simulation model, also establishes a contract between participating federates, according to their characteristics, functions, and localization. For the execution of a model, federates must implement services defined in a HLA Interface Specification, in order to interact with the RTI. This constraint limits the reuse of federates that have been built for executing on other operating system platforms or even according to other implementations of the RTI. The DCB principles of independence and preservation of elements (federates in HLA) overcome these restrictions.

Recent research on the HLA standard has promoted the opening of new possibilities for distributed simulation, including the use of heterogeneous models [1]. Motivated by the inherent complexity of heterogeneous simulation, these studies look for alternatives or extensions to HLA, in order to meet the requirements of heterogeneous models. One of the main issues is the adaptation between parts (federates in HLA) that have incompatible interfaces or have been developed with different languages.

Some works have concentrated on issues regarding the interoperability between different implementations of the RTI, since the RTI interface is not standardized and federates developed for a given implementation may not be interoperable with other ones. There are proposals both for a standard RTI interface and for the creation of RTI-to-RTI bridges (translators) [9]. The standardization of the RTI interface may increase the difficulty in the reuse of existing federates that use incompatible interfaces, since the federate must be modified. The use of RTI-to-RTI bridges makes interoperability dependent on the existence of a bridge. Thus, for each different RTI, all other RTIs must have a corresponding bridge.

Proposals have also considered the combined utilization of different standards (for instance CORBA and UML) as alternative solutions for the interoperability between HLA implementations [22]. Evolution has been achieved in this perspective, for instance in the mapping from RTI to CORBA/IDL (Interface Description Language). However, the characteristics of alternative standards, such as CORBA and UML, which have been developed for other domains, have to be taken into account when applied in combination with simulation mechanisms, since new problems arise. As a consequence, the standards or the associated general-purpose tools may need to be adapted to the simulation domain.

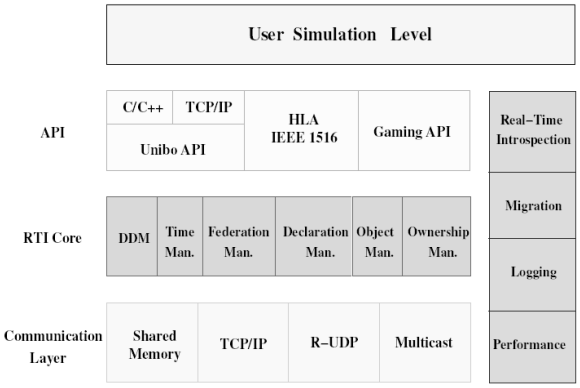

Figure 1: ARTIS architecture [2]

Recent efforts have been focused on alternatives seeking more flexibility in handling distributed models targeting at heterogeneous components (or federates). An example is the ARTIS (Advanced RTI System) framework [2]. Based on the HLA standard, it proposes to support the execution of distributed and parallel simulations and the cooperation between federates. Differently from HLA, ARTIS implements an API layer for federate integration allocated between the federate and the RTI. This way, instead of executing explicit calls to the primitive RTI, the federate interacts with a compatible API. This policy allows the federates that were developed to the implementation of an RTI to be reused in another one. Even so, the integration of a federate depends on the availability of an API that is compatible with the federate interface. Figure 1 presents the structure in ARTIS layers.

The WESE environment [15] supports the distributed execution of simulation models. The construction of a model has as first requirement the specification of components (or elements) in a language called SSL (System Specification Language). This description is used to entail the model to the internal modules of WESE which are responsible for some functions such as communication and data handling. A simulation manager (SM) executes this task. The SM does not

communicate directly with the components, but with a repository manager (RM). The RM recognizes the component interface through object stubs. The object stubs contain the component attributes and information about machine requirements for simulation and about the component behavior.

Approaches in heterogeneous distributed simulation are usually focused on solving problems coming from particular application domains. As a consequence, there is a tendency in emphasizing only certain features, while other ones are not considered. This can be observed in different solutions that are targeted to the embedded systems domain.

In multi-language approaches [10], for instance, the integration of new components is restricted to languages that are supported by the tools, while hybrid synchronization and component preservation are not considered. A similar situation can be observed in the JavaCAD tool [6]. It allows to instantiate IP (intellectual property) components from multiple remote providers within a distributed simulation model. However, although JavaCAD targets at heterogeneous models, components may only be connected to the model through a particular JavaCAD client, which handles the communication. A still higher specificity regarding the development of heterogeneous models is found in commercial tools that are targeted at electronic systems, such as from Coware [4], Cadence [3], Mentor Graphics [18], and Synopsys [20].

In general, existing environments and tools are targeted at given application domains and do not consider generality as a main requirement. It may be hard to redesign solutions that are proprietary or based on a given set of domain-specific requirements in order to support new features of a different domain. It is thus desirable the development of a general-purpose environment that simultaneously addresses independence and reuse of components (federates in HLA, elements in DCB), hybrid synchronization, and transparent distributed simulation. These principles form the basis of the DCB architecture, which shows greater flexibility and generality in the distributed execution of heterogeneous models than other current approaches.

3 DCB

DCB has been developed as an intermediate layer between elements and the communication medium, offering support to the distributed execution of heterogeneous simulation models. It offers generic mechanisms for communication and synchronization between heterogeneous elements, with four main goals: the physical distribution of the elements; the independence between the elements; the encapsulation of mechanisms for time management, adaptation between elements, and communication; and hybrid synchronization. The encapsulation of these mechanisms inside the DCB infrastructure allows a greater independence of elements, thus meeting the requirements of interoperability and interchangeability (substitution of elements) in the cooperation between heterogeneous elements.

DCB does not require that elements make explicit calls to a simulation backplane for distribution and synchronization operations. As opposed to other proprietary solutions, DCB does not impose proprietary standards for data exchange. Therefore, DCB reduces the need for modifications in the implementation of elements to be integrated.

The DCB infrastructure is general-purpose and is not affected by particular simulators or sub-models to be integrated into a heterogeneous model. These DCB features make the integration of already existing elements much easier and more flexible. DCB does not impose restrictions on its use in different simulation domains. The independence between elements and the encapsulation of the mechanisms for managing the distribution and heterogeneity are not based on requirements of a particular domain.

As opposed to general-purpose middleware solutions, like CORBA, which deal with language interoperability and distribution, DCB also considers these requirements but is specifically oriented towards distributed heterogeneous simulation.

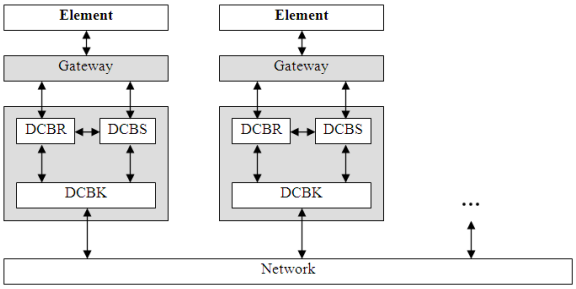

Figure 2: DCB architecture

In the DCB approach, a heterogeneous model is composed of autonomous and distributed elements. Elements may be described with different languages and/or simulated by any simulators. In order to participate in a heterogeneous model, an element must have a publicly available interface.

This means that attributes of the interface need to be visible and controllable from outside. By controlling these attributes, DCB may configure the way how cooperation between elements is performed and may implement mechanisms for the automatic configuration of a heterogeneous model, including the required adaptation between distinct element interfaces and implementation languages. This way, internal aspects of an element do not impact its integration into a model. If elements have been already validated, as expected, the system designer may worry only about their integration in the model.

Figure 2 shows the DCB architecture. It is composed by four main modules: DCBSender (DCBS), DCBReceiver (DCBR), DCBKernel (DCBK), and gateway. The gateway's main task is to handle the element interfaces, while the other modules handle synchronization, data management, and cooperation. The rest of this section presents the mechanisms that implement these services and the role of each module of the DCB infrastructure in the distributed execution of heterogeneous models.

Model configuration

DCB manipulates information ruling cooperation between the elements of a model. This information is provided by a mechanism of configuration. Before the execution of any simulation task that enables DCB to manage cooperation between the elements of a model, configurable data structures are read separately from the model and DCB modules. This action is called 'configuration work'. For each one of the elements a particular data structure must be created. The next section presents Tangram, an environment that is responsible for this task. These are some examples of information related to the configuration of each element:

- Identification and type of input and output attributes;
- Destination of each output attribute;
- Type of synchronization;
- Node hosting the element for execution.

Such information describes the interface of the element used by the Gateway (DCB module) to exchange data resulting from the execution of the model. The description of interface attributes and the other configuration information are specified by the designer of the model during its construction. Aspects of generality and flexibility of DCB have in the model configuration strategy an important basis for the preservation of DCB internal modules. They avoid the need of alterations in the implementation of these modules for the execution of different models.

Time Management

Each element has its own Local Virtual Time (LVT), which defines a temporal ordering on events within the element. The DCB kernel also maintains a unique Global Virtual Time (GVT) for synchronous elements and another one for asynchronous [7] ones. This global time is used to build a global ordering on events from different elements. In order to implement this ordering, the current DCB prototype implements a special-purpose element, named TimeMgrGVT. When any element tries to advance its LVT, a corresponding message is automatically sent to TimeMgrGVT, which then re-computes the GVT and, if it is advanced, communicates its new value to all other elements. The elements' LVTs cannot be advanced beyond the GVT in the synchronous mode.

DCB also supports the inclusion of elements that do not consider a local time for event execution (untimed elements). Untimed elements do not require consistency in time for output data, but just to maintain an order in the source element output in the respective destination. The cooperation between elements with distinct modes of time advancement is called hybrid synchronization. This feature of DCB is based on the fact that a better cost/benefit relationship can be achieved by hybrid models when compared to pure untimed, synchronous, or asynchronous models. Since a synchronous element cannot advance its internal time beyond the GVT, it may remain idle while waiting other elements to advance, even if it does not depend on events coming from them. In asynchronous elements, in turn, the independent time advancement by an element may optimize the simulation time, but causality constraints between elements may be violated (violations of LCC-Local Causality Constraints), so that rollback may be required. By combining both types of synchronization, DCB explores their advantages simultaneously. In [23] positive aspects in the use of hybrid synchronization are highlighted, with the consequent increase of complexity in the model execution.

Gateway

Gateways adapt element interfaces to the heterogeneous model and also implement adapters between programming languages used in the implementation of each element, if necessary. In order to participate in a heterogeneous model, an element must have its interface publicly available (see Section 4) and update its interface attributes by using a single gateway method:

Gateway.UpdateAttribute ("attribute name", value, timestamp);

This rule also applies to the element's LVT, which must be available as an interface attribute and controllable from the outside. This is a requirement for the integration of any simulator or model into a DCB heterogeneous model. For this, it uses configuration information. The gateway recognizes native methods of the element interface to send data to it. The gateway is also responsible for data type conversions, when needed. If the element encapsulates an IP component that is remotely simulated, the gateway and the component will be located in different hosts.

The gateway has four main responsibilities regarding the administration of the interface between an element and internal modules of DCB:

1 – Monitoring the updating of values of the output attributes of the element interface;

2 – Data type conversion according to the configuration of the heterogeneous model (translation);

3 – Semantically converting data according to the configuration of the model (e.g. when data available in an attribute at the source element must be split up and delivered in two different attributes at the destination);

4 – Updating input attributes of the element when requested by the internal modules of DCB.

These characteristics of the gateway are requirements for the DCB to support the independence of elements (preservation of element interfaces and internal implementations) and the encapsulation of mechanisms for simulation administration. The interface adaption requires changes just in the gateway code.

DCBReceiver

The DCBReceiver (DCBR) module receives messages from DCBK (executing communication operations) and sends them to the elements it represents, via gateway, according to the simulation time (synchronization function). It is DCBR's responsibility the consistent use of configuration information in the handling of messages. Among these, it must redirect the message contents to the correct input attributes of the destination element (decoding). The input attributes (in the destination) for each of the output attributes (from the source) are defined in the configuration of the heterogeneous model.

The DCBR does not execute remote operations. Messages are received by the primitives of communication of DCBK. In the decoding process of received messages, the DCBR identifies the destination attribute and its type, among other control information, and maintains the message in a waiting list. The message will only be sent to the element when there is consistency with regard to the model time. If two or more messages have the same timestamp, the arrival order at the destination is respected. In some situations, in which the source element of a message is configured as 'notime', the contents of the message is sent to the destination element without taking into consideration the time, however, in the same order as in the source.

DCBSender

The DCBSender (DCBS) has the general goal of providing the necessary mechanisms to receive requests of messages transmitted by the element via its gateway, codifying data, and sending the messages to the destination using the DCBK communication services. The gateway is sensible to value alterations in output attributes of the element it represents.

Multiple output attributes, of a single or multiple source elements, can send data to the same input attribute in a destination element. In these conditions, DCBS maintains the principles of synchronization while passing messages to the respective input attribute in the destination element. Keeping the history of sent messages is part of the DCBS task. This history is used for returning to a safe state when the simulation faces the occurrence of errors caused by violations of LCC [7]. The time rollback to a safe state is a policy foreseen only for elements configured for the execution in the asynchronous time mode.

While sending messages coming from elements defined as 'notime', DCBS assigns a value to the time control field of the message, which does not need to be consistent in time. DCBR identifies such messages, which are immediately forwarded to the element (ordered according to the sending element).

DCBKernel

The DCBKernel (DCBK) implements the primitives of communication between elements of a heterogeneous model. These primitives use configuration informations of elements to manage the exchange of messages. Each output attribute has an input attribute in the destine defined in the individual element configuration.

DCBK disregards the content of messages and does not interfere in local actions of their treatment. It neither differentiates messages generated by elements from administrative messages created by DCBS and DCBR. The separation between network operations and the administrative actions of simulation gives flexibility to the treatment of messages in the distributed operations.

DCBK recognizes automatically (from configuration information) if a destination element is located in the same node or in a remote node. If it is located in the same node, DCB implements communication through a direct call of specific methods. If the element is in a different node, then sockets are used for the remote communication. The DCBS and DCBR modules are not involved with communication issues, which are maintained only by DCBK.

DCBK is also responsible for the storage of a consistent. This module is the only one to execute operations related to communication with remote nodes, making it suitable for extensions aiming at safety and fault tolerance policies (e.g. in case of node falling or regarding IP component information protection).

4 Tangram modeling environment

The Tangram modeling environment presents functionalities for the practice of specific activities for the heterogenous models. By adding functionalities to generate information about the elements of a model, to the interface administration and to the modeling (assistance in the creation of links between interface attributes of elements) the Tangram incorporates the principles of generality and flexibility defined in the DCB architecture. These principles interfere positively for new functionalities. In order to extend the DCB architecture presented in Section 3, some specific characteristics were added to enhance the implementation aspects. These characteristics are inspired in specific purpose environments for the simulation of Embedded Systems (co-simulation). Thus, heterogeneous models are built by instantiating and interconnecting elements that are stored in local repositories, which are hierarchically organized. These repositories may also contain references to elements that are only remotely available.

The services offered by Tangram modeling environment can be identified in the cooperated work of 4 main

modules as it is shown in Figure 3: a *graphical modeling tool*, an *import assistant*, an *adapting functional interfaces*, and a *configuration tool*. Section 5.1 comments on the aspects of implementation of the modeling environment prototype separately for each one of these modules.

In order to store an element in a local repository and later instantiate it in a heterogeneous model, the environment supplies conditions for the designer to construct an explicit declaration of the element's interface. Because of this public interface, Tangram does not need to know internal details of an element in order to integrate it into a heterogeneous model.

The interface of an element may have several *access points*. Each access point may have several alternative definitions, corresponding to levels of different detailing for the element, corresponding to different stages of specification of the model of system being analyzed and designed. In a preliminary model where implementation details are not specified yet, the access point may be defined as a single port offering a collection of high-level access *methods*, with input and output parameters. In a level of posterior specification, in which the detailing of interface implementation is important, the access point may be defined as a bundle of *ports*, each having its own data type. This task is done in a working area supplied by the *graphical modeling tool*.

The interface specification of a local or remote element results in an IPD (IP Description) file. The definition includes the element name, the location of the code describing the element behavior (maybe an URL for a remote element), the language used for describing the element behavior, and the icon that represents the element in the interface modeling tool.

The generation of the heterogeneous model is performed in two main steps. In the first one, elements are instantiated but only an identification of access points is introduced. This way, two elements may be interconnected without being needed, in this stage, any treatment of possible incompatibilities in data formats or protocols used by the source and destination elements. In a second stage, the interface ports or methods contained in the access points are exposed. If interfaces of interconnected elements match each other, interface ports or methods may be interconnected. If interfaces of interconnected elements do not match exactly, still, an adaptation is necessary.

Incompatible interfaces are indicated in the user interface environment. Adaptations can be done in the element code (*adapting functional interfaces*) or, in case of IP components for which the source code is not available, *wrappers* must be built. Since the interface of an IP element is publicly known, wrappers may be built without knowledge of internal details of the element. Wrapper construction is based on the IPD files describing the element interfaces.

The last stage before execution of the model is attended by the *configuration tool* that performs two main tasks. Firstly, it generates XML files that are used for the configuration of the DCBS and DCBR modules (or internal modules). These files are dynamically read during the initialization of the internal modules. This avoids their recompilation for each heterogeneous model. The second task is the compilation of the gateways. The input for these tasks is obtained from the XML specification of the heterogeneous model, generated by the graphical modeling tool.

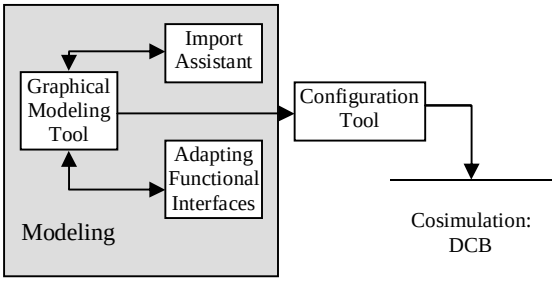

Figure 3: Tangram Modules

Besides the heterogeneous model configuration, the XML specification is also used to determine the mode of communication between local and remote elements. For local elements, message exchange is implemented through direct function calls that do not use network services, such as sockets, which are necessary to interconnect remote elements. These different communication mechanisms help improve the simulation performance.

5 Tangram/DCB prototype

This section presents the Tangram/DCB prototype developed for the simulation of heterogeneous models in the domain of the Embedded Systems (ESs) project. For this reason, the implementation shows some aspects which were influenced by this domain's characteristics, however without impairment of the generality principles defined in the DCB architecture.

In the implementation of the prototype, the cooperation between the elements described in different levels of abstraction, relevant to the ESs project, can be observed mainly in the treatment of access points in the modeling environment. In DCB, the concern with the construction of templates for the cooperation between elements described in different modeling and hardware description language also reveals dependency on ESs characteristics.

In DCB the concern with the templates is related to the reutilization of IP components, an important characteristic in the ESs domain. Project techniques oriented to the reutilization contribute to reduce cost (e.g. failure, time) in the process of these systems project. This has motivated the development of methodology for the project and reutilization of IPs [19].

5.1 Tangram modeling environment implementation

This section presents details of implementation of modules making the Tangram modeling environment.

IP-based graphical modeling

The graphical modeling tool allows the instantiation of local or remote IP elements that are available in the repositories and creates an XML description of a heterogeneous model, including all information on elements and their interconnections.

In the first step of the generation of the heterogeneous model, the elements are instantiated and their interface access points are shown, as illustrated in Figure 4, which shows a screenshot of the modeling tool during the definition of the heterogeneous model for the case study presented in (Section 6). In this example, where only the identification of the access points (AP) is shown, the access point 'rc_update' of the *GPSAlert* element receives a display update request sent by the *GPS* element through its access point 'sd_req_up', while the access point 'sd_coord' sends new coordinates to the access point 'coord_in' of *Display_Driver*. Details of interface ports or methods inside the access points are hidden at this abstraction level. In this example, two elements are connected in a high level of abstraction, as it can be seen in the connection between the APs 'sd_req_up' and 'rc_update' as well as between the APs 'sd_coord' and 'coord_in'.

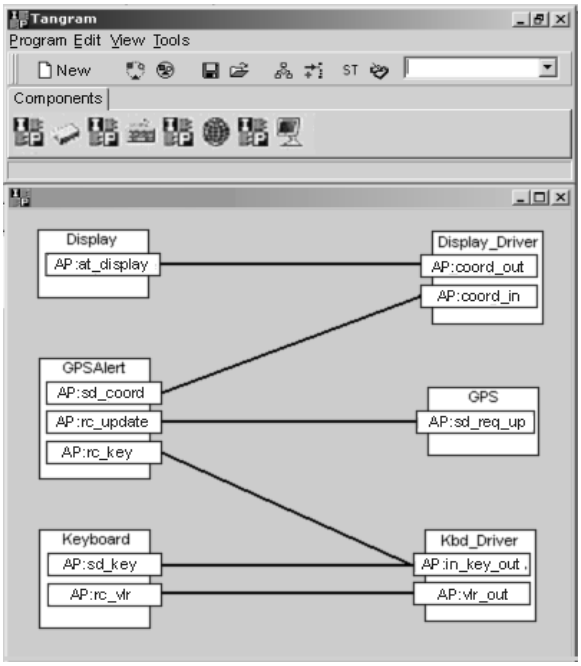

Figure 4: Instantiating and connecting elements through access points

After the connection between the access points, the environment allows the visualization of details of gates and methods in each one of the access points created (second step). In this stage the designer can identify compatible and incompatible interfaces. Incompatible interfaces are indicated in the user interface through different colors attached to the connections between the access points, as well as through pop-up messages.

Import Assistant

The import assistant helps the user in locating remote elements and, if desired and possible, retrieving and storing them in the local repositories. The import assistant é pré-requisito para a tarefa de modelagem no graphical modeling. The information about IPs maintained by this module is essential in the process of creating elements for the construction of models.

Adapting functional interfaces

The environment does not interfere in the adaptation of translation of incompatible interfaces when it can be done in the element itself. This type of alteration is dependable on the level of access to the element. For the adaptation using wrappers, the environment offers general templates that allow the creation of wrappers configured from the IPD files. The designer must manually complete the wrapper with its adaptation to the interface protocol of the other element to which this element is connected. Tangram also helps the user to reuse and specialize previous wrappers. A repository of reusable wrappers for connecting various elements to given bus or interface standards may be thus created.

From the graphical model, the tool automatically generates an XML file that contains all information on the heterogeneous model and its elements. This file is completely hidden from the user. It is passed to the configuration module that will generate the required data structures for the simulation.

Configuration Module

The Configuration Module implements the service that creates a structure of individual data in an XML file for each model element from the modeling information. The use of this module must precede the execution of the model **over** DCB.

5.2 DCB implementation

The mechanisms implemented in the modules which support the distributed execution of heterogeneous models are shown in Figure 5. In order to extend the DCB architecture presented in Section 3, some specific characteristics of this implementation were added. The inclusion of these characteristics is partially inspired in properties that were identified in the ESs co-simulation context. Nevertheless, the accuracy in relation to the DCB architecture is kept unspoiled.

In this implementation the Gateway module was divided in two parts and the DCBMthread was added. Being so, multiple elements of the same node can be executed over the same JVM. This decision of implementation reduces the use of memory and allows that exchange of messages between remote elements, enhancing performance (important in the ESs co-simulation).

The configuration tool described in the previous section, apart from generating the XML with individual

configuration of each element, it also generates automatically the corresponding gateways (by configuring library templates) for particular languages / simulators and access methods. A template is a code skeleton that is automatically filled by a configuration tool, introduced in Section 5.1. In this implementation there are specific templates of the ESs domain to specify interface with tools of hardware description. Current templates correspond to the following alternatives:

• An element with a Java interface can directly communicate with the gateway by function calls and parameter passing, since the gateway (as the other DCB modules) is also implemented in Java;

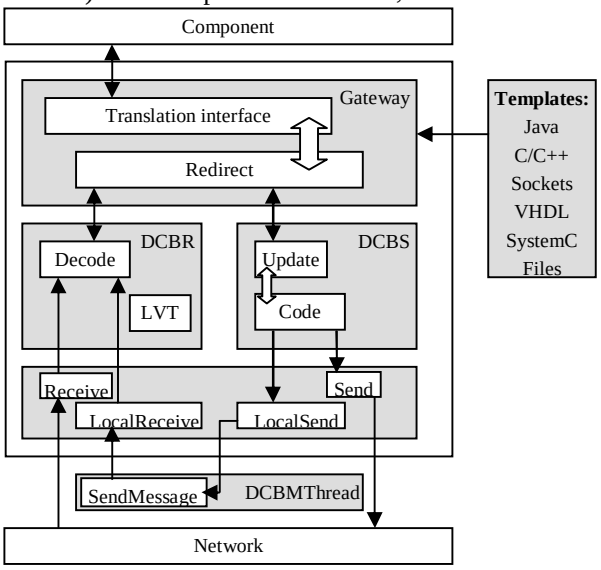

Figure 5: DCB Implementation

• An element with a C/C++ interface, when implemented as a dynamic link library (.dll), can be directly loaded by the gateway. Communication is performed by routines that access native code offered by JNI (Java Native Interface);

• For an element with a C/C++ interface, whose source code is available, a template is added to the code and invokes JVM (Java Virtual Machine), thus allowing function calls through JNI to Java objects in the gateway;

• A VHDL element is integrated by Modelsim APIs (MTI.h library) that provide socket connections;

• A SystemC element for which a header and a pre-compiled object source are available may be integrated into heterogeneous models by using three auxiliary entities: an adapter module, a simulation driver, and the gateway. Each one of the ports of a SystemC element is connected through a signal to a corresponding port of an adapter module. These connections are used by a simulation driver, which updates output attributes of the element and looks for values of the input attributes through the gateway. The simulation driver uses JNI calls for communicating with the gateway; and

• An element that is located in a remote host can communicate with the gateway by interprocess mechanisms such as sockets. In remote cooperation, only the communication functions that are implemented in the DCB kernel are automatically adjusted for using adequate communication primitives. Actions implemented in the DCBS/DCBR modules and in the gateway are not affected.

For the configuration actions, the implementation has a general interface, identified in Figure 5 as 'Application DCB', which has as initial operation the reading of configuration information of the element (generated by the modeling environment) and uses the internal data structures that will be used by the DCB afterward for the simulation administration. Some examples of this information are: attributes identification, element localization, time synchronization type used by the element, among others.

The DCBR module has a Decode block that, when receiving a message from other elements according to the configuration, maintains an organized list based on the timestamp of the messages. They are forwarded to the Gateway module synchronizing he timestamp and the GVT (if the source element is temporized), maintaining the consistency with the sending order (if the source element is not temporized). The calculation of LVT is done in this module by means of permanent element TimeMgrGVT (as described in Section 3). For this, the DCBR of each element has an administrative message that communicates the TimeMgrGVT of all alteration requests of LVT. Whenever the TimeMgrLVT detects an alteration of GBT, the DCBK of all elements is informed.

The Code block of DCBS module, after receiving a request resulting from the communication between the update module of DCBS and the Gateway, forwards the received values to the destination element (identifying the respective input attribute) specified in the configuration file. Before sending the message, this is coded to an appropriate format of DCB aiming facilitating the administration work, such as translation. As the messages actions ensuring order, synchronization, translation, and others, are executed by DCB in the destination, the DCBS solicits its sending on request. Mechanisms of message retention are implemented in this module just to seize problems. For instance, the DCBS does not send messages with inconsistent timestamp in relation to GVT (for synchronous elements in time) generating a warning.

The DCBK module, apart from the in force GVT register, does not practice operations of simulation administration. It implements the communication operations that take into consideration the requirements generated in the DCBS Code module, either between local elements (direct call to the LocalReceive and LocalSend methods) or remote ones (use of sockets –

Send and Receive). IN the local communication the DCBK uses the SendMessage method available in the DCBMthread module and in communication with remote elements network services are used. This is the only module practicing remote operations.
The administration of messages which are sent to local nodes also have the support of redirect module. This module is located before the transaltion interface implemented in the Gateway to cooperate with the element.
The implementation was done in Java language due to two main reasons, the flexibility to execute operations in the web and the ease of portability [21]. Java characteristics in relation to the web incorporate relevant **capability** (regarding portability and communication) in the development of new approaches to modeling and simulation. The portability issue facilitates the cooperation of heterogeneous elements in terms of executing platform, a plausible situation in ESs projects. For example, in the study of the case presented in Section 6, an element described in VHDL and executed with ModelSim over SunOS was added to the model with elements described in C++ for Windows. Thus, expect for Gateway interface adaptations, no alteration was needed in DCB modules or in the model.

6 Case study

This section illustrates the capabilities of the DCB and Tangram prototypes by partially describing the design of a portable GPS-Alert terminal. It receives GPS coordinates, compares them with user-defined key points previously stored in memory, and alerts the user about a point that is approaching, by displaying its identification.

High-level functional model

Figure 6 shows a first functional model of the system, which does not imply architectural definitions. It includes 4 elements, all of them described in Java.

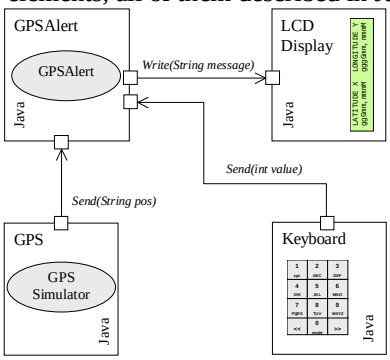

Figure 6: Functional model of GPS-Alert system

Computation and communication are described at a high abstraction level. For communication, a transaction-level model is built, using primitives like send, receive, read, and write. Element interfaces in the figure are represented only by the access points. Since element interfaces exactly match each other, wrappers are not required. GPS-Alert is the main system element. It stores key point coordinates, receives GPS data, compares coordinates, and communicates with keyboard and display. Keyboard and Display are abstract Java models of the real peripherals. GPS-Simulator, which has only validation purposes, simulates the generation of a sequence of coordinates, by reading them from a text file. An XML heterogeneous model specification is generated by the modeling tool. Gateways can be automatically generated by configuring Java templates. This first model is completely homogeneous.

Architectural-level model

The model is then refined to an architectural definition, as shown in Figure 7. Three IP elements are reused: a Java microcontroller [12], for implementing the main *GPS-Alert* functions, and keyboard and display drivers, to connect the microcontroller to real peripherals. In this model, while the drivers are locally available at the designer's site, the microcontroller is a third-party IP model that is remotely simulated at a provider's site. Considering the DCB co-simulation capabilities, any element may be locally or remotely simulated, without any impact in the heterogeneous model's functionality and in the internal description of the elements.
In this architectural model, the *GPS-Alert* element description, still written in Java, now mixes an abstract specification of the computation with a refined communication at RT level. As previously, computation is described by the *GPS-Alert* functionality.

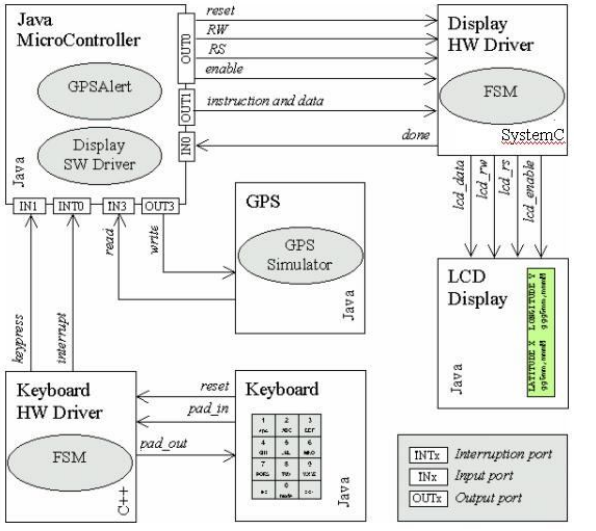

Figure 7: Architectural model of GPS-Alert system

Communication, however, considers the real microcontroller interface, consisting of I/O ports and interrupt signals. The same mixed-level modeling in Java is used for the *GPS-Simulator*, *Keyboard*, and *Display* elements. Figure 7 shows the interface attributes contained in the element access points.

Kbd-Driver is described in C++. *Display-Driver* is described in SystemC and implements a proprietary interface. A heterogeneous (Java/C++/SystemC) and distributed heterogeneous model is thus built. Gateways and internal modules are generated by configuring Java, C++, and SystemC templates. For C++ and SystemC elements, besides the gateways, also the C++ code accessing the JNI functions is generated. Table 1 shows the size (in source code lines) of elements and respective gateways.

Table 1: Size of elements and gateways

Element	Element Language	Element size (lines)	Size of gateway generated (lines)
TimeMgrGVT	Java	81	140
GPS Alert	Java	349	159
GPS Simulator	Java	107	120
LCD Display	Java	398	138
Display driver	SystemC	269	218 * **
Keyboard	Java	208	152
Keyboard drv	C++	309	243 *

* includes the C++ code that gives access to JNI
** includes the simulation driver (75 lines) and the complementary module (11 lines)

Even for much more complex elements, their gateways will still have reduced sizes, since their sizes are only proportional to the number of interface signals of the respective IP components. The internal modules for all elements have always the same size: 172 lines of Java code for FA and 206 lines for DCBA. The DCB kernel also has a constant size of 227 lines of Java code. TimeMgrGVT, also with a constant size of 221 lines, is automatically inserted into the heterogeneous model. It is responsible for the overall synchronization, as explained in Section 4.

Simulation performance

In order to make a concrete comparison of simulation times between different models, we observe the time consumed by a complete screen update, performed through a series of messages sent to the *Display* element. This activity starts when a first update message is sent and ends when the last message has been processed by the *Display* element. In the high-level functional model, messages are sent by the *GPS-Alert* element, while in the architectural model the messages are sent by the *Display-Driver* element.

The number of messages for a display update is larger in the architectural model because of the model refinement at a lower abstraction level. In this model, 282 messages are required for a complete display update, while in the functional model only 42 messages are required. In both cases, 9 control messages, sent by DCB and related to synchronization between the elements, are required.

Both models have been initially simulated with the *Display* element located in the same network node as the respective origin element. In a second step, the *Display* element has been moved to a distinct node, connected by a 100 Mbps network adapter. Both nodes have been completely isolated from the remaining network. In the functional model, the mean time to update the display has been 60 ms in the local simulation and 320 ms in the distributed one. In the architectural model, the local execution took 79 ms and the distributed one 304 ms.

It can be observed that the architectural model is only 19 ms (or 31.7%) slower than the functional one, when we consider only local simulation. This overhead is due both to the refined description of the communication and to the language adaptation (all elements in the functional model are described in Java, while the *Display-Driver* element in the architectural model is described in SystemC).

The distributed execution, in turn, is faster in the architectural model when compared to the functional one, even if the simulation is performed at a lower abstraction level. This apparently odd behavior is due to the way the *Display-Driver* element is implemented in the architectural model. It executes an infinite loop, constantly monitoring its interface attributes, thus consuming a large simulation time. In the distributed execution, this element has been allocated to a separate node. In this way, the processing power of this node is entirely devoted to this element, which is not executed concurrently with the other elements anymore.

7 Final remarks and future work

In this article we introduce an infrastructure for the integration of heterogeneous elements called Distributed Cosimulation Backbone (DCB). It is not targeted towards simulation performance. Its main goals are the preservation of elements, adaptation of heterogeneous interfaces, and distributed simulation. With the objective of offering resources for the construction of models and the adaptation of interfaces, the article also introduces the Tangram as a modeling and configuration environment aiming the execution over DCB.

Tangram and DCB do not impose severe rules on the description of the elements' communications, which could limit the reuse of already existing heterogeneous elements (including IP components). Beyond the reuse we also enhance the construction of new elements with tools/languages best adapting to the objectives of the model being constructed.

The management of distribution and communication between elements that are located at different sites and described with different languages is entirely encapsulated within gateways and internal modules that are automatically generated and kept independent from the element's code.

This principle of DCB operation is fundamental for the purposes of integrity preservation of elements and the adaptation of interfaces. This feature distinguishes

Tangram from other general-purpose distributed communication solutions.

Some studies of case were presented, developed in the domain of the embedded systems project, in two different levels of abstraction: high-level functional model and architectural-level model. In the first level all the elements are described in the same language and they do not incorporate architecture details of the real system. In the second level three more elements were integrated into the first version of the model; two of them described in different language, incorporating some architectural details. Despite the changes, there was no impact over the functionality of the model or over the internal description of the elements previously integrated into the model.

In order to enhance simulation performance, simulation code generation will consider in the future dedicated implementations for gateways and internal modules in two special cases: non-distributed models and homogeneous models. Future work will also consider capabilities for elements classification and search, to be added to the modeling environment.

References

[1]Bononi, L. et al. Concurrent Replication of Parallel and Distributed Simulations. In *Proceedings of the PADS'05*,Monterey,CA,USA, June 2005.

[2]Bononi, L., Bracuto, M., D'Angelo, G. AND Donatiello, L. ARTIS: a Parallel and Distributed Simulation Middleware for Performance Evaluation. In *Proceedings of ISCIS'2004*, Antalya, Turkey, October 2004.

[3]CADENCE. 2008. *Incisive™ unified Simulator*. Available at <http://www.cadence.com>.

[4]COWARE. 2008. Available at <http://www.coware.com>.

[5]Dahmann, J. S. High Level Architecture for Simulation. In Proceedings *of International Workshop On Distributed Interactive Simulation And Real-Time Applications-Dis-Rt*, Eilat, Israel, 1997.

[6]Dalpasso, M., Bogliolo, A. AND BENINI, L. Virtual simulation of distributed IP-based designs. In *Proceedings of 1999 Conference On Design Automation Conference*, New Orleans, USA, p.50-55, 1999.

[7]Elnozahy, M. et al. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Technical Report CMUCS99148, Department of Computer Science*, Carnegie Mellon University, 1999.

[8]Girod, L. et al. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of 2004 Embedded Networked Sensor Systems Conference*, Baltimore, USA, p.201-213, 2004.

[9]Granowetter, L. RTI Interoperability Issues – API Standards, Wire Standards, and RTI Bridges. In *Proceedings of 2003 European Simulation Interoperability Workshop*, Stockholm, Sweden, 2003.

[10]Hessel, F. et al. MCI: Multilanguage Distributed Cosimulation Tool. In *F.J.Rammig (ed.), Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, 1999.

[11]IEEE 1516. IEEE Standard for Modeling and Simulation High-Level Architecture (HLA), 2001.

[12]Ito, S., Carro, L. and Jacobi, R. Making Java Work for Microcontroller Applications. In *IEEE Design & Test of Computers*, Sept/Oct, 2001.

[13]Lee, E. A. Computing for Embedded Systems. In *Proceedings of IEEE Instrumentation And Measurement Tecnhology Conference*, Budapest, Hungary, 2001.

[14]McBratney, A. et al. Future Directions of Precision Agriculture. In *Precision Agriculture, Springer Netherlands Publisher,* vl. 6, n. 1, February, 2005.

[15]Rao, D. M., Chernyakhovsky, V. AND Wilsey, P. A. WESE: A Web-based Environment for Systems Engineering. In *Proceedings of International Conference On Web-Based Modelling & Simulation*, Orlando, Florida, 2000.

[16]Reynolds, P. F. Jr. Heterogeneous Distributed Simulation. In *Proceedings of 1998 Winter Simulation Conference*, p.206-209, 1998.

[17]Schubert, K. Improvements in functional simulation addressing challenges in large, distributed industry projects. In *Proceedings of 2003 Conference On Design Automation,* Anaheim, p.11-14, 2003.

[18]SEAMLESS CVT. 2008. *Mentor Graphics.* Available at <http://www.mentor.com >.

[19]Seepold, R. Reuse of IP and virtual components. In *Proceedings of 1999 Design, Automation, And Test In Europe*, Munich, Germany, 1999.

[20]SYNOPSYS. 2008. Available at <http://www.synopsys.com>.

[21]Teo, Y. M.; Ng, Y. K.; Onggo, B. S. S. Conservative Simulation using Distributed-Shared Memory. In *Proceedings of Workshop On Parallel And Distributed Simulation,* Washington-D.C, USA, 2002.

[22]Tolk, A. Avoiding another Green Elephant – A Proposal for the Next Generation HLA based on the Model Driven Architecture. In *Proceedings of Simulation Interoperability Workshop,* Orlando, Florida, USA, 2002.

[23]Yoo, S., Choi, K. AND Ha, D. S. Performance Improvement of Geographically Distributed Co-simulação by Hierarchically Grouped Messages. *IEEE Transactions on VLSI Systems, New York,* v.8, n.5, p.100-104, 2000.