Tangram: Virtual Integration of IP Components in a Distributed Cosimulation Environment

Bráulio Adriano de Mello
Universidade Regional Integrada do Alto Uruguai e das Missões

Uilian Rafael Feijó Souza, Josué Klafke Sperb, and Flávio R. Wagner
Universidade Federal do Rio Grande do Sul

IP reuse is essential in embedded SoC design, but IP components can use different modeling languages and present heterogeneous interfaces. To efficiently integrate these heterogeneous components, the Tangram environment supports the remote evaluation of IP components, implementing the virtual integration of these components into distributed cosimulation models.

A LARGE VARIETY OF LANGUAGES exist to model the different types of hardware and software components in modern embedded SoCs, as well as to deal with various models of computation (MoCs) and abstraction levels. Although single languages have been proposed as standards, covering a wide modeling range, the most common situation is the development of heterogeneous models, where different languages describe components. Cosimulation tools handling heterogeneous models become necessary for validating these designs.

The reuse of IP components is essential for coping with very tight time-to-market constraints. IP protection issues might impose restrictions on the access to virtual component models. An IP component might not be downloadable from the provider's site before its purchase, or it might require a simulator that is available only at the provider's site. The evaluation of an IP component might thus require its "virtual" integration into a cosimulation model, so that it can be remotely simulated at the provider's site while the SoC designer accesses its functionality only through its interface.

The heterogeneity of IP components is complicated by the fact that their functional interfaces might not directly match each other. Direct matching would be possible if all components use the same interface standard. The diversity of IP providers, together with the search for best-of-class components, however, can result in designs requiring the integration of components with heterogeneous interfaces.

A general approach to the evaluation of IP components must therefore provide a simultaneous and consistent solution to three major problems:

- the interoperability among different modeling languages,
- the adaptation of the heterogeneous functional interfaces of IP components, and
- the virtual integration of heterogeneous IP components into distributed cosimulation models.

Current approaches to cosimulation do not simultaneously address these three issues. Tangram is a cosimulation environment offering general-purpose, innovative solutions for these three problems.

We named it after an ancient type of Chinese puzzle that has as its goal the building of a regular figure by fitting together pieces of various sizes and shapes.

Tangram works on top of a general-purpose infrastructure for distributed simulation, called the Distributed Cosimulation Backbone (DCB). It offers mechanisms for the easy integration of existing IP components into cosimulation models and for the use of new modeling languages.

0740-7475/05/$20.00 © 2005 IEEE Copublished by the IEEE CS and the IEEE CASS

The Tangram and DCB mechanisms are not restricted to a given set of modeling languages. Instead, they are based on concepts from High-Level Architecture (HLA), an IEEE standard for distributed simulation that allows for the easy integration of new languages.[1] However, we have adapted HLA concepts to an IP reuse scenario, in which components cannot depend on the cosimulation mechanisms.

Tangram encapsulates adaptation layers for communication and synchronization among heterogeneous and remote components within the DCB cosimulation infrastructure, so that the code for components becomes completely independent from these DCB mechanisms. Besides, Tangram automatically generates these layers, requiring only a knowledge of the components' interfaces. This promotes IP reuse by avoiding the code adaptations sometimes necessary for integrating a component into a cosimulation model.

The integration of an IP component into a cosimulation model in Tangram follows a stack of layers with different functions. The various layers support the adaptation of functional interfaces and languages, and distribution and synchronization. This separation of concerns lets Tangram enhance the reuse of adapted components for synthesis purposes, achieves an easier integration of new languages, and can implement a more efficient cosimulation of models that do not have remote components.

Current situation

As just mentioned, Tangram addresses problems in three major areas.

Language interoperability

Usual approaches to language interoperability are based on proprietary or restricted solutions. Commercial cosimulation tools are restricted to given combinations of SystemC, VHDL, Verilog, C, C++, instruction set simulators, or native compiled code. A more general solution, the Multilanguage Cosimulation Interface (MCI), automatically builds a multilanguage cosimulation model, based on a proprietary, distributed cosimulation backplane.[2] MCI simulations, however, handle only untimed models. Ptolemy is a framework offering a set of classes that support interoperability between objects following different models of computation.[3]

This and similar approaches, however, do not directly address the interoperability among common languages, such as C, Java, and VHDL. In contrast, Tangram offers a general-purpose mechanism for language integration, just as MCI, but also supports timed models.

IP integration

Designers can easily interconnect IP components into a design if the component interfaces follow a bus standard, such as AMBA,[4] or a core standard, such as OCP (http://www.ocpip.org). The integration of components with heterogeneous interfaces, however, requires hardware and/or software wrappers. A design tool can automatically generate hardware wrappers if interfaces are formally described.[5] COSY maps high-level communications between components to communication schemes, for which there is a library of hardware-software wrappers.[6] The Roses tool generates hardware-software wrappers by composing basic modules corresponding to given component types and communication structures.[7] From a SystemC specification, Roses generates wrappers for both cosimulation and synthesis, using the same wrapper architecture.

Tangram is not restricted to a given bus or core standard and can adapt any interface, provided the designer specifies them. Unlike all the other approaches, Tangram also handles language interoperability for cosimulation.

Distributed simulation

Researchers have proposed distributed, heterogeneous cosimulation techniques for the Web-based evaluation of IP components. In JavaCAD, distributed simulation follows a client-server approach and provides IP protection for remote components, but it requires the description of all components in Java.[8] In HLA, components communicate by calling runtime infrastructure (RTI) functions. However, this process restricts the integration to already-available IP components. Such approaches suppose that IP components, from a functional point of view, have consistent interface protocols that can connect directly to each other, without requiring application wrappers.

Tangram is strongly inspired by the HLA standard, so we did not restrict it to particular design languages. However, Tangram does not require that components make explicit calls to RTI functions. As opposed to all the other approaches to distributed simulation, Tangram also considers the adaptation of incompatible functional interfaces among components.

Tangram/DCB overview

We aimed Tangram at the virtual integration of heterogeneous IP components into distributed cosimulation models. It offers resources for adapting the functional interfaces of IP components, if they do not match directly. Tangram also implements interoper-

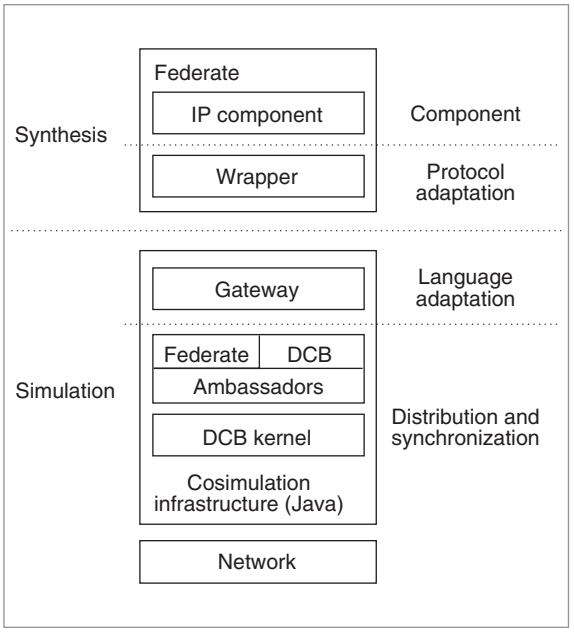

Figure 1. Tangram integration layers.

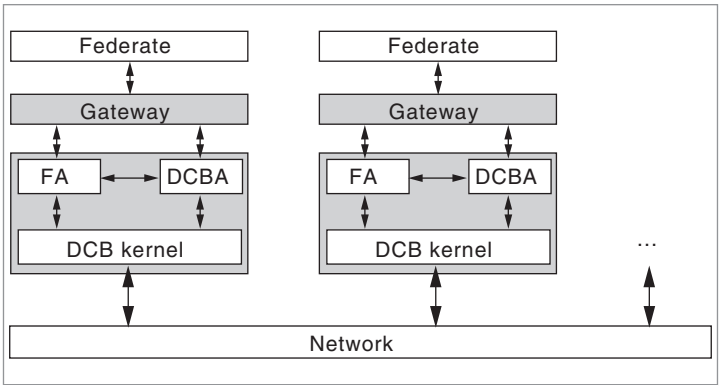

Figure 2. DCB architecture.

ability between different design languages.

We based Tangram cosimulation on DCB.[9] DCB, in turn, uses simulation principles from the HLA standard. HLA was initially conceived within the community of distributed and interactive simulation, considering special needs observable in military training. Although some conceptual definitions in DCB are similar to those in HLA, we proposed DCB as an environment that is completely independent from HLA.

DCB is a simulation-specific coordination layer for supporting the distributed simulation of heterogeneous systems. It offers generic mechanisms for communication and synchronization among heterogeneous components. The encapsulation of these mechanisms inside the DCB infrastructure allows a greater independence of components. As opposed to the HLA standard, DCB does not require that components make explicit calls to an RTI; it also does not require the use of HLA mechanisms for distribution and synchronization.

As opposed to other proprietary solutions, DCB does not impose proprietary standards for data exchange. Therefore, DCB reduces the need to modify the component implementations to support integration. The DCB infrastructure is general purpose and remains unaffected by which particular simulators or submodels the cosimulation effort must integrate into a federation. These DCB features make the integration of already-existing IP components much easier and more flexible.

Generic middleware solutions, such as the Common Object Request Broker Architecture, deal with language interoperability and distribution. In contrast, we specifically oriented Tangram toward distributed cosimulation, including synchronization capabilities. We also developed features targeted at embedded-SoC designs that support the integration of heterogeneous IP components.

In the Tangram approach, as in HLA, a cosimulation model is a federation, composed of autonomous and distributed federates. Federates can have descriptions in different languages and/or simulations by any simulator. A federate encapsulates an IP component's code. To participate in a federation, a federate must only have a publicly available interface. This means that the interface's attributes must be visible and controllable from outside. By controlling these attributes, DCB can configure the way federates cooperate and implement mechanisms for the automatic configuration of a cosimulation model. This way, internal aspects of an IP component do not impact its integration into a federation. If IP components are already validated, as expected, the SoC designer need only worry about their integration through the federation.

Figure 1 shows the Tangram layers for component integration. Components for interconnection can present incompatible interface protocols or have various abstraction levels. Component wrappers implement protocol- and abstraction-level adaptation. Wrappers are usually necessary for both cosimulation and synthesis, so the description of the federation, including the interface wrappers, are synthesizable by appropriate tools.

For cosimulation, Tangram automatically generates gateways, which adapt functional interfaces and languages. It also generates ambassadors, which connect federates to the DCB kernel. Ambassadors, together with the DCB kernel, implement distribution and synchronization among various components.

Figure 2 shows the DCB architecture. Federates don't

need to invoke DCB communication primitives for sending data. Instead, a federate transfers output data to its gateway, which communicates with the DCB kernel through a pair of ambassadors.

Although gateways can imply some implementation effort, they avoid more costly modifications both in the communication and synchronization infrastructure (DCB kernel and ambassadors) and in the code of simulators and federates. Gateways depend on the respective federates, but Tangram automatically configures them when it builds a new federation. Ambassadors and the DCB kernel, in turn, have a fixed code that does not depend on federates.

Via this stack of layers, Tangram implements a separation of concerns. Each layer has a different function:

- the wrappers adapt the functional interfaces,
- the gateways adapt the languages, and
- the DCB kernel and the ambassadors implement distribution and synchronization.

This separation of concerns achieves several goals. First, it enhances the reuse of components for synthesis purposes by adapting the functional interfaces. Second, it becomes easier to integrate new languages because the gateways provide encapsulation. Lastly, a more-efficient cosimulation of models that do not have remote components is possible, via the implementation of specialized ambassadors and DCB kernels for nondistributed models.

Modeling environment

The Tangram modeling environment has four main modules: a graphical modeling tool, an IP repository, an import assistant, and a configuration tool. Users can build cosimulation models by instantiating and interconnecting components stored in local, hierarchically organized repositories. These repositories can also contain references to components that are only remotely available. An import assistant helps the user in locating remote components and retrieving and storing them in the local repositories.

Interface specification

To store a federate in a local repository and later instantiate it in a cosimulation model, a designer must explicitly declare its interface through an import assistant. Because of this public interface, Tangram does not need to know internal details of a federate to integrate it into a federation.

A component's interface can have several access points. Each access point can have several alternative definitions, corresponding to various abstraction levels. At RTL, for instance, the access point might be a bundle of ports, each having its own data type. At higher levels of abstraction—such as service, message, and transaction—the access point might be a collection of access methods, with input and output parameters. There are no predefined names for abstraction levels.

The interface specification of a local or remote IP component results in an IP description (IPD) file, which defines the interface by means of an XML configuration. The definition includes the component name, the location of the code describing the component behavior (for example, a URL for a remote component), the language used for describing the component behavior, and the icon that represents the component in the graphical modeling tool.

IP-based graphical modeling

The graphical modeling tool allows the instantiation of local or remote IP components that are available in repositories. It also creates an XML description of a federation, including all information on federates and their interconnections. Generating the cosimulation model requires two main steps.

In the first step, designers instantiate components, but only show their interface access points. Figure 3 shows a screenshot of the modeling tool during the definition of the federation for the case study that we present later. This first step shows only the identification of the access points. In this example, access point rc_update of the GPSAlert federate receives a display-update request sent by the global positioning system (GPS) federate through its access point, sd_req_up, while 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. This way, designers can interconnect two components even if their interfaces at a given abstraction level do not match exactly.

A second step exposes the interface ports or methods contained in the access points (at a selected abstraction level). If the interfaces of interconnected components match each other, designers can interconnect interface ports or methods.

From the graphical model, the tool automatically generates an XML file that contains all the information on the federation and its federates. The Tangram completely hides this file from the user. It passes the file to

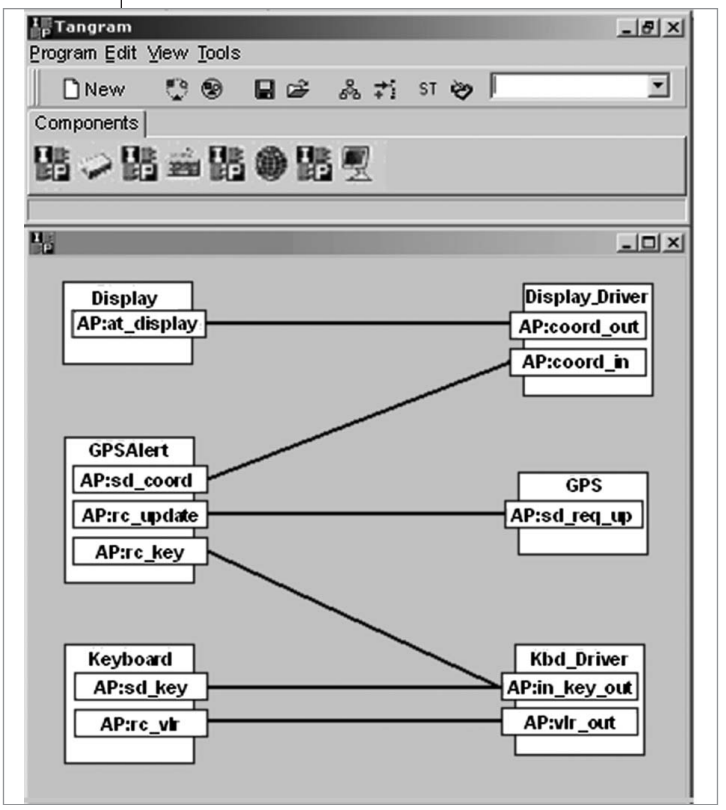

Figure 3. Instantiating and connecting components through access points (AP).

the configuration tool that will generate the required data structures for the simulation.

Adapting functional interfaces

If the interfaces of interconnected components do not match exactly, an adaptation is necessary. The user interface of the graphical modeling tool indicates incompatible interfaces through pop-up messages and by using different colors attached to the connections between the access points. The user can implement adaptation on the IP component source code, if it is available.

IP components without their source code require wrappers. Because the interface of an IP component is public knowledge, designers can build wrappers without knowledge of the component's internal details. Wrapper construction is based on the IPD files describing the component interfaces. Wrappers can also follow general templates, and Tangram can partially configure them from the IPD files. The designer must manually complete the wrapper with its adaptation to the interface protocol of the connected component.

Tangram also helps designers to reuse and customize previous wrappers. Designers can thus create a repository of reusable wrappers for connecting various components to component interfaces with given bus or core standards.

Configuration tool

The configuration tool performs two main tasks. First, it generates XML files that Tangram uses to configure the ambassadors, dynamically reading the files during the ambassadors' initialization. Doing so avoids recompiling the ambassadors for each federation. The second task is the compilation of the gateways, based on predefined templates, as we explain in the next section. The input for these tasks comes from the federation's XML specification, which the graphical modeling tool generates.

Besides the federation configuration, Tangram also uses the XML specification to determine the mode of communication between local and remote federates. Local federates use message exchange through direct function calls that do not use network services, such as sockets; this scheme reserves those resources to interconnect remote federates. These different communication mechanisms help improve the simulation performance.

Distributed Cosimulation Backbone

This section presents the role of each internal module of the DCB infrastructure (kernel, gateway, and ambassadors) in the distributed execution of heterogeneous cosimulation models.

DCB kernel

The DCB kernel manages synchronization and data exchanges between simulators. DCB supports a hybrid synchronization that allows a combination of synchronous, asynchronous, and untimed federates. Each federate has its own local virtual time (LVT), which defines a temporal ordering on events within the federate. The DCB kernel also maintains a unique global virtual time (GVT) for synchronous federates and another one for asynchronous federates. DCB uses this global time to build a global ordering on events from different federates.

To implement this ordering, the current DCB prototype implements a special-purpose federate, FedGVT. When any federate tries to advance its LVT, a corresponding message automatically goes to FedGVT, which then recomputes the GVT and, if it is advanced, communicates its new value to all other federates. In synchronous mode, the federates' LVTs cannot advance

beyond the GVT. In asynchronous mode, the global time does not restrict a federate's time advancement, but the simulator and the federates must support the return to a previous safe state (rollback).[10] This is necessary because the kernel occasionally receives events with time stamps that are past its current local time.

DCB also supports the inclusion of federates that do not consider a local time for event execution (untimed federates). In this case, DCB maintains an LVT that the ambassadors control; such an LVT thus remains transparent to the federate.

The literature calls the cooperation between federates with distinct modes of time advancement *hybrid synchronization*. We base this DCB feature on the fact that a better cost-benefit relationship is achievable by hybrid models when compared to purely synchronous or asynchronous models. Because a synchronous federate cannot advance its internal time beyond GVT, it might remain idle while waiting for other federates to advance, even if it does not depend on events coming from them. In asynchronous federates, in turn, the independent time advancement by a federate might optimize the simulation time but violate causality constraints among federates, thereby requiring rollback. By combining both types of synchronization, DCB explores their advantages simultaneously.

Gateways

Gateways adapt federates' interfaces to the federation and also implement adapters between languages, if necessary. To participate in a federation, a federate must have its interface publicly available (as a following case study describes) and update its interface attributes by using a single gateway method:

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

This rule also applies to the federate'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 federation. The gateway recognizes native methods of the federate's interface to send data to it. The gateway is also responsible for data type conversions, when needed. If the federate encapsulates a remotely simulated IP component, the gateway and the federate will reside in different hosts.

Gateways are automatically generated by configuring library templates for particular languages, simulators, and access methods. A template is a code skeleton that a configuration tool automatically fills, as the case study will describe. Current templates correspond to the following alternatives:

- A federate with a Java interface can directly communicate with the gateway by function calls and parameter passing because the gateway is also implemented in Java (as are the other DCB modules).
- A federate with a C or C++ interface, when implemented as a dynamic link library, can be directly loaded by the gateway. Communication occurs via routines that access native code offered by the Java Native Interface (JNI).
- For a federate with a C or C++ interface, whose source code is available, Tangram adds a template to the code that invokes a Java virtual machine. This permits function calls through the JNI to Java objects in the gateway.
- A VHDL federate uses the Modelsim APIs (MTI.h library) for integration. These APIs provide socket connections.
- A SystemC federate for which a header and a precompiled object source are available requires three auxiliary entities for integration into cosimulation models. These entities are an adapter module, a simulation driver, and the gateway. Each one of a SystemC federate's ports connects to a corresponding port of an adapter module via a signal. A simulation driver uses these connections to update a federate's output attributes and look for values of the input attributes through the gateway. The simulation driver uses JNI calls for communicating with the gateway.
- A federate located in a remote host can communicate with the gateway by interprocess mechanisms such as sockets. Tangram automatically adjusts only the communication functions implemented in the DCB kernel, such that they use adequate communication primitives. This does not affect actions implemented in the ambassadors and in the gateway.

Ambassadors

The federate ambassador (FA) registers a federate's name and type, and the properties of its interface attributes. FA is responsible for handling synchronization aspects and for storing in an input queue the attribute values that are received from the DCB kernel.

The DCB ambassador (DCBA) stores information on interconnections between federates. It manages the

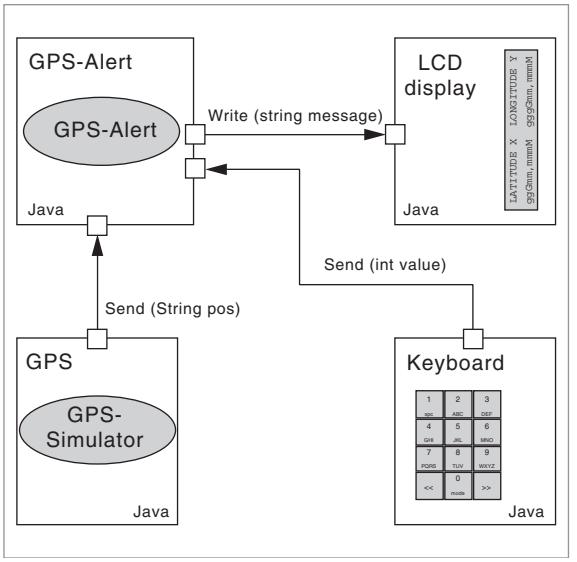

Figure 4. Functional model of the GPS-Alert system.

however, must check if the source federate has ownership over the destination attribute. If positive, DCBA removes the message from the output queue, and sends it to the DCB kernel. If not, DCBA will request ownership, which the DCB kernel will give, depending on a set of rules. These rules, in turn, depend on the federates' LVTs and on the GVT.

Case study

Here, we illustrate Tangram capabilities 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 an approaching point by displaying its identification.

High-level functional model

Figure 4 shows a first functional model of the system; this model does not imply any architectural definitions. It includes four Java components; it describes computation and communication at a high abstraction level. For communication, we built a transaction-level model, using primitives such as send, receive, read, and write. In the figure, access points represent component interfaces. Because the component interfaces match each other exactly, the components do not require wrappers.

GPS-Alert is the main system component. It stores key point coordinates, receives GPS data, compares coordinates, and communicates with the 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. The modeling tool generates an XML federation specification, and the configuration of Java templates automatically generates gateways. This first model is completely homogeneous.

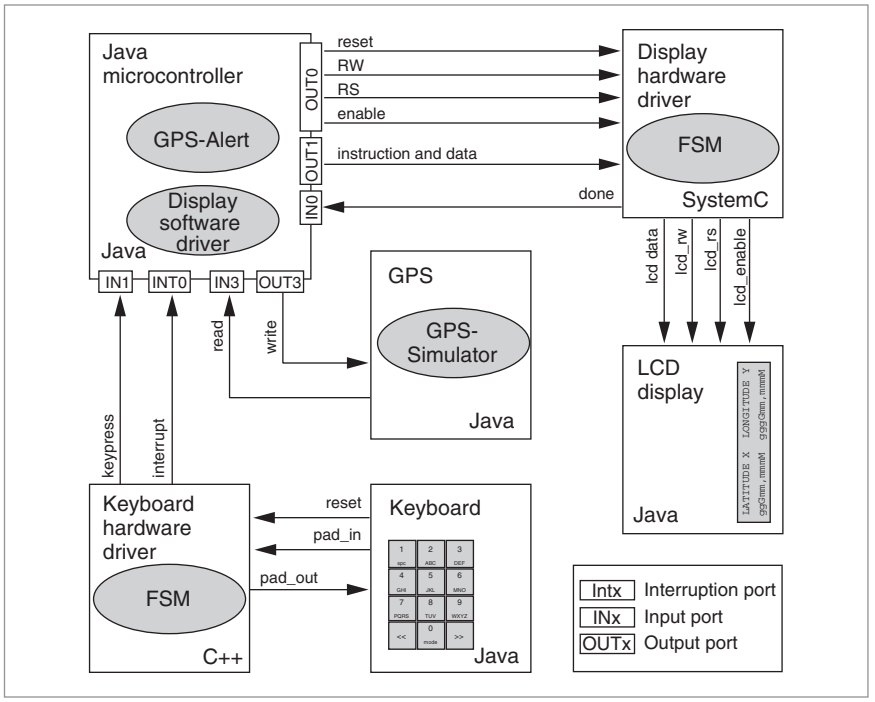

Figure 5. Architectural model of the GPS-Alert system.

attributes' ownership as well as the history of values (checkpoints) that the federation exchanges. It retains this information for rollback purposes, in the case of asynchronous federates. When a federate sends a message through an output attribute, this message is initially stored in an output queue in DCBA. From the federate's perspective, the message has been sent. DCB,

Architectural-level model

We next refine the model into the architectural definition in Figure 5. It reuses three IP components: a Java microcontroller for implementing the main GPS-Alert functions,[11] and keyboard and display drivers, to connect the microcontroller to real peripherals. In this

model, although the drivers are locally available at the designer's site, the microcontroller is a third-party IP model that requires remote simulation at a provider's site. Considering the DCB cosimulation capabilities, it's possible to locally or remotely simulate any federate without impact to the federation's functionality or to the federates' internal descriptions.

Table 1. Size of federates and gateways.

Federate	Federate language	Federate size (LOC)	Gateway size (LOC)
FedGVT	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 driver	C++	309	243*

* Includes the C++ code for JNI access.

** Includes the simulation driver (75 LOC) and the adapter module (11 LOC).

In this architectural model, the GPS-Alert component description, still written in Java, now mixes an abstract specification of the computation with a refined communication at RTL. As before, the GPS-Alert functionality describes the computation. Communication, however, considers the real microcontroller interface, consisting of I/O ports and interrupt signals. We use the same mixed-level modeling in Java for the GPS-Simulator, keyboard, and display components. Figure 5 shows the interface attributes contained in the component access points.

We describe the keyboard driver in C++; the display driver is in SystemC and implements a proprietary interface. This makes the federation heterogeneous (containing Java, C++, and SystemC components) and distributed. Configuring Java, C++, and SystemC templates generates gateways and ambassadors. For C++ and SystemC federates, besides the gateways, we also generate the C++ code accessing the JNI functions. Table 1 shows the size (in lines of code, or LOC) for federates and their respective gateways.

Even much more complex federates will have gateways of reduced sizes, because a gateway's size is only proportional to the number of interface signals passing through it from its respective IP components. The ambassadors for all federates always have the same size: 172 LOC in Java for the FA and 206 LOC for the DCBA. The DCB kernel also has a constant size of 227 LOC in Java. FedGVT, also with a constant size of 221 LOC, automatically becomes part of the federation. It is responsible for the overall synchronization, as explained earlier.

Simulation performance

To concretely compare simulation times among different models, we observe the time consumed by a complete screen update, performed through a series of messages sent to the display federate. This activity starts with a first update message and ends when the display federate processes the last message. In the high-level functional model, the GPS-Alert federate sends messages; in the architectural model, the display driver sends the messages.

The number of messages for a display update is larger in the architectural model because the model refinement is at a lower abstraction level. In this model, a complete display update requires 282 messages; the same action requires only 42 messages in the functional model. Both cases require nine control messages, sent by DCB and related to synchronization among the federates.

We have initially simulated these models with the display federate located in the same network node as the respective origin federate. In a second step, we moved each display federate to a distinct node, completely isolated from the remaining network and connected by a 100-Mbps network adapter. In the functional model, the mean time to update the display had 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 execution took 304 ms.

We 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 comes from the refined description of the communication and to the language adaptation (all the components in the functional model use Java descriptions, whereas the display driver component in the architectural model uses a SystemC description).

The distributed execution, in turn, is faster in the architectural model than in the functional model, even if we perform the simulation at a lower abstraction level.

This apparently odd behavior arises from the way we implement the display-driver federate in the architectural model. It executes an infinite loop, constantly monitoring its interface attributes, thus consuming a large simulation time. The distributed execution allocates this federate to a separate node. In this way, this node's processing power is entirely devoted to this federate, which no longer executes concurrently with the other federates.

WE DO NOT TARGET Tangram toward simulation performance. Rather, its main goals are language interoperability, the adaptation of heterogeneous interfaces, and distributed simulation. Together, these features allow the virtual integration of heterogeneous IP components into cosimulation models, resulting in rapid IP evaluation while maintaining IP protection.

Tangram and DCB do not impose severe rules on the description of the components' communications, which could limit the reuse of already-existing, heterogeneous, IP components. Tangram entirely encapsulates the management of distribution and communication among components—located at different sites and described with different languages—within gateways and ambassadors that it automatically generates and keeps independent of the federates' code. This feature distinguishes Tangram from other general-purpose distributed communication solutions.

To enhance simulation performance, we will, in the future, consider simulation code generation that uses dedicated implementations of gateways and ambassadors for two special cases: nondistributed and homogeneous models. Future work will also consider capabilities for IP classification and search within the modeling environment. ■

Acknowledgments

We gratefully acknowledge the support from the Brazilian funding agencies CNPq and Capes. Carlos Arthur Lisboa developed the original Java model of the GPS-Alert terminal. Further development of this model, targeted at the Tangram environment, had valuable help from Daniel Barden, Lucio O.M. Rech, Fabio Wronski, and Eduardo W. Brião.

■ References

1. IEEE Std. 1516.1-2000, *IEEE Standard for Modeling and Simulation (M&S) High-Level Architecture (HLA)—Federate Interface Specification*, IEEE, 2000.
2. F. Hessel et al., "MCI: Multilanguage Distributed Cosimulation Tool," *Distributed and Parallel Embedded Systems*, F.J. Rammig, ed., Kluwer Academic Publishers, 1999.
3. J. Davis II et al., "Overview of the Ptolemy Project," technical memorandum UCB/ERL M01/11, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley, Mar. 2001.
4. *AMBA Specification Rev2.0*, ARM Ltd., Mar. 1999.
5. R. Passerone, J.A. Rowson, and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," *Proc. 35th Design Automation Conf.*, ACM Press, 1998, pp. 8-13.
6. J.-Y. Brunel et al., "COSY Communication IP's," *Proc. 37th Design Automation Conf.*, ACM Press, 2000, pp. 406-409.
7. W. Cesario et al., "Component-Based Design Approach for Multicore SoCs," *Proc. 39th Design Automation Conf.*, ACM Press, 2002, pp. 789-794.
8. M. Dalpasso, A. Bogliolo, and L. Benini, "Virtual Simulation of Distributed IP-Based Designs," *Proc. 36th Design Automation Conf.*, ACM Press, 1999, pp. 50-55.
9. B.A. Mello and F.R. Wagner, "A Distributed Co-simulation Backbone," M. Robert et al., eds., *SOC Design Methodologies*, Kluwer Academic Publishers, 2002.
10. M. Elnozahy et al., *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*, tech. report CMUCS99148, Dept. of Computer Science, Carnegie Mellon Univ., Sept. 1999.
11. S. Ito, L. Carro, and R. Jacobi, "Making Java Work for Microcontroller Applications," *IEEE Design & Test of Computers*, vol. 18, no. 5, Sept.-Oct. 2001, pp. 100-110.

Bráulio Adriano de Mello is a professor in the Engineering and Computer Science Department of the Universidade Regional Integrada do Alto Uruguai e das Missões (URI), Santo Ângelo, Brazil. His research interests include systems modeling and heterogeneous simulation. Mello has a BS in computer science from the Universidade de Passo Fundo and a PhD in computer science from UFRGS. He is a member of the Brazilian Computer Society (SBC).

Uilian Rafael Feijó Souza is a graduate student at UFRGS. His research interests include the architecture and design of embedded systems, the distributed cosimulation of heterogeneous systems, IP reuse, and component-

based systems development. Souza has a BS in computer science from Universidade Federal de Pelotas.

Josué Klafke Sperb is a technician at the Fundação de Economia e Estatística (FEE), Porto Alegre, Brazil. His research interests include distributed and heterogeneous cosimulation. Sperb has a BS in computer science from the Universidade de Santa Cruz do Sul and an MSc in computer science from UFRGS.

Flávio R. Wagner is a professor at the Computer Science Institute of UFRGS. His research interests include embedded-systems modeling, design, and cosimulation. Wagner has a BS in electrical engineering from UFRGS and a PhD in computer science from the University of Kaiserslautern. He is a member of the IEEE Computer Society, ACM SIGDA, SBC, and the Brazilian Microelectronics Society (SBMicro).

■ Direct questions and comments about this article to Bráulio Adriano de Mello; Rua Universidade das Missões, 464; Santo Ângelo – RS; 98802-470, Brazil; bmello@urisan.tche.br.