UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOSUÉ KLAFKE SPERB

Geração de Modelos de Co-Simulação Distribuída para a Arquitetura DCB

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, novembro de 2003.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Sperb, Josué Klafke

Geração de Modelos de Co-Simulação Distribuída para a Arquitetura DCB / Josué Klafke Sperb – Porto Alegre: Programa de Pós-Graduação em Computação, 2003.

103 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientador: Flávio Rech Wagner.

1. Simulação distribuída. 2. Co-design. 3. Co-simulação. 4. Co-simulação distribuída e heterogênea. 5. Reuso. 6. Interoperabilidade. 7. Propriedade intelectual. 8. HLA. 9. DCB. 10. Java. 11. JNI. I. Wagner, Flávio Rech. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Wrana Maria Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitora Adjunta de Pós-Graduação: Prof.ª Jocélia Grazia
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS 4
LISTA DE FIGURAS 6
LISTA DE TABELAS 8
RESUMO 9
ABSTRACT 10
1 INTRODUÇÃO 11
1.1 Objetivos 12
1.2 Contribuições 13
1.3 Estrutura do Texto 14
2 SIMULAÇÃO DISTRIBUÍDA 15
2.1 Simulação Paralela e Distribuída (PADS) 15
2.1.1 Sincronização Conservativa 16
2.1.2 Sincronização Otimista 16
2.2 Simulação Distribuída Interativa (DIS) 17
2.3 HLA (*High Level Architecture*) 18
2.3.1 Especificação de interface HLA (HLA *Interface Specification*) 20
2.3.2 Modelos de Objetos HLA (HLA *Object Models*) 21
2.3.3 Regras HLA (HLA *Rules*) 22
2.3.4 Abordagens para integração de federados através da interface HLA 23
2.3.5 Interoperabilidade no padrão HLA 24
3 CO-DESIGN E CO-SIMULAÇÃO 26
3.1 Co-design 26
3.1.1 Linguagens de Especificação e Projeto 27
3.2 Co-Simulação 28
3.2.1 Ambientes de co-simulação 31
3.3 Trabalhos relacionados 33
3.3.1 Ptolemy 33

3.3.2 IPChinook 33
3.3.3 DISCOE 34
3.3.4 Ambiente S^3E^2S 34
3.3.6 MCI (Multilanguage Distributed Co-Simulation Tool) 34
3.3.7 ROSES 35
4 ARQUITETURA DCB 36
4.1 Distributed Co-Simulation Backbone (DCB) 40
4.2 Embaixadores 41
4.2.1 Embaixador do DCB (EDCB) 42
4.2.2 Embaixador do Federado (EF) 44
4.3 Gateway 46
4.4 Inicialização, controle e acompanhamento de experimentos de co-simulação utilizando a arquitetura DCB 48
5 IMPLEMENTAÇÃO 50
5.1 Arquitetura DCB 50
5.2 DCB (*Distributed Co-Simulation Backbone*) 52
5.3 Embaixador do DCB (EDCB) 53
5.4 Embaixador do Federado (EF) 54
5.5 Templates do Gateway 55
5.5.1 *Template* Java 57
5.5.2 Template Sockets 58
5.5.3 *Template* JNI 60
5.6 Ferramenta para configuração dos embaixadores e geração do *gateway* 63
6 ESTUDO DE CASO 66
6.1 Experimento 1 67
6.2 Experimento 2 68
6.3 Experimento 3 71
6.4 Conclusões do Estudo de Caso 73
7 TRABALHOS FUTUROS 74
7.1 Otimização de desempenho 74
7.2 Comunicação de simuladores VHDL com *gateways* da arquitetura DCB 75
7.2.1 Modelo mestre-escravo 76
7.2.2 Modelo distribuído 77
7.3 Comunicação de modelos SystemC com *gateways* da arquitetura DCB 77
8 CONCLUSÃO 80

REFERÊNCIAS 82
ANEXO A TEMPLATE JAVA 91
ANEXO B TEMPLATE JNI 93
ANEXO C EMBAIXADOR DO FEDERADO (EF) 97
ANEXO D EMBAIXADOR DO DCB (EDCB) 99
ANEXO E ARQUIVO DE CONFIGURAÇÃO PARA O FEDERADO DISPLAY NO EXPERIMENTO 2 102
ANEXO F MODELO DA FEDERAÇÃO (XML) UTILIZADO NO EXPERIMENTO 1 103

LISTA DE ABREVIATURAS

ALSP	*Aggregate Level Simulation Protocol*
API	*Application Programming Interface*
ASIC	*Application-Specific Integrated Circuit*
CLI	*C Language Interface*
CMB	*Chandy-Misra-Bryant*
CORBA	*Common Object Request Broker Architecture*
DCB	*Distributed Co-Simulation Backbone*
DCOM	*Distributed Component Object Model*
DIF	*Data Interchange Format*
DIS	*Distributed Interactive Simulation*
DISCOE	*DIstributed Simulation COllaborative Environment*
DLL	*Dynamic Link Library*
DMSO	*US Defense Modeling and Simulation Office*
DoD	*Department of Defense*
EDCB	*Embaixador DCB*
EF	*Embaixador Federado*
FLI	*Foreign Language Interface*
FOM	*Federate Object Model*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
GPS	*Global Position System*
GVT	*Global Virtual Time*
HDL	*Hardware Description Language*
HLA	*High Level Architecture*
IDL	*Interface Description Language*
IP	*Intellectual Property*
IPC	*Inter-Process Communication*

ISS	*Instruction-Set Simulator*
JNI	*Java Native Interface*
JVM	*Java Virtual Machine*
LCC	*Local Causality Constraint*
LCD	*Liquid Crystal Display*
LP	*Logical Process*
LVT	*Local Virtual Time*
MCI	*Multilanguage Distributed Co-Simulation Tool*
OMT	*Object Model Template*
PADS	*Parallel and Distributed Simulation*
RT	*Register Transfer*
RTI	*Run-Time Interface*
RTL	*Register Transfer Level*
S^3E^2S	*Specification, Simulation, and Synthesis of Embedded Electronic Systems*
SOM	*Simulation Object Model*
TCP	*Transmission Control Protocol*
VCI	*VHDL-C Interface*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
XML	*eXtensible Markup Language*

LISTA DE FIGURAS

Figura 2.1: Visão funcional de uma federação HLA 20
Figura 2.2: O paradigma dos embaixadores na interface HLA 21
Figura 3.1: Hardware / Software *co-design* 26
Figura 4.1: Integração de sistemas heterogêneos 36
Figura 4.2: Arquitetura DCB (*Distributed Co-Simulation Backbone*) 37
Figura 4.3: Visão funcional da arquitetura DCB 38
Figura 4.4: Detalhes internos da arquitetura DCB 39
Figura 4.5: Arquitetura da camada DCB 40
Figura 4.6: Arquitetura do Embaixador do DCB (EDCB) 42
Figura 4.7: Ligação entre atributos de federados (configuração da federação) 44
Figura 4.8: Arquitetura do Embaixador do Federado (EF) 44
Figura 4.9: Arquitetura do *Gateway* 46
Figura 4.10: Formas para implementar a comunicação entre o *gateway* e o federado 48
Figura 5.1: Protótipo da arquitetura DCB 51
Figura 5.2: Protótipo da camada DCB 52
Figura 5.3: Modelo para comunicação remota implementado no DCB 53
Figura 5.4: Protótipo do embaixador do DCB (EDCB) 53
Figura 5.5: Protótipo do embaixador do federado (EF) 55
Figura 5.6: Trecho de código do *template* Java 55
Figura 5.7: Detalhes internos do *template* Java 57
Figura 5.8: Detalhes do funcionamento do conversor de protocolos 58
Figura 5.9: Detalhes internos do *template sockets* 58
Figura 5.10: Especificação formal (DTD) das mensagens trocadas entre federado e *gateway* 59
Figura 5.11: Requisitos para um federado comunicar-se através de *sockets* 59
Figura 5.12: Exemplo de funcionamento da JNI (*Java Native Interface*) 60
Figura 5.13: Detalhes internos do *template* JNI (*Java Native Interface*) 61
Figura 5.14: Esquema de tomada de decisão da função *UpdateAttribute* 61
Figura 5.15: Abordagem alternativa para integração de federados 62
Figura 5.16: Exemplo de integração de um federado C/C++ 62

Figura 5.17: Ferramenta de configuração... 63
Figura 5.18: Especificação formal (DTD) do arquivo gerado pela ferramenta de modelagem.. 64
Figura 5.19: Detalhes internos da arquitetura da ferramenta de configuração.......................... 64
Figura 5.20: Especificação formal (DTD) do arquivo de configuração dos embaixadores........ 65
Figura 6.1: GPS Alerta modelado em um alto-nível de abstração 67
Figura 6.2: Detalhes do micro-controlador FemtoJava 69
Figura 6.3: Modelo do GPSAlerta utilizado no experimento 2... 69
Figura 6.4: Modelo GPSAlerta utilizado no experimento 3.. 71

LISTA DE TABELAS

Tabela 6.1: Detalhes do experimento 1 68
Tabela 6.2: Detalhes do experimento 2 70
Tabela 6.3: Detalhes do experimento 3 72

RESUMO

O aumento na complexidade dos sistemas embarcados, compostos por partes de hardware e software, aliado às pressões do mercado que exige novos produtos em prazos cada vez menores, tem levado projetistas a considerar a possibilidade de construir sistemas a partir da integração de componentes já existentes e previamente validados. Esses componentes podem ter sido desenvolvidos por diferentes equipes ou por terceiros e muitas vezes são projetados utilizando diferentes metodologias, linguagens e/ou níveis de abstração. Essa heterogeneidade torna complexo o processo de integração e validação de componentes, que normalmente é realizado através de simulação.

O presente trabalho especifica mecanismos genéricos e extensíveis que oferecem suporte à cooperação entre componentes heterogêneos em um ambiente de simulação distribuída, sem impor padrões proprietários para formatos de dados e para a descrição do comportamento e interface dos componentes. Esses mecanismos são baseados na arquitetura DCB (*Distributed Co-Simulation Backbone*), voltada para co-simulação distribuída e heterogênea e inspirada nos conceitos de federado (componente de simulação) e federação (conjunto de componentes) que são definidos pelo HLA (*High Level Architecture*), um padrão de interoperabilidade para simulações distribuídas.

Para dar suporte à co-simulação distribuída e heterogênea, esse trabalho descreve mecanismos que são responsáveis pelas tarefas de cooperação e distribuição, chamados de embaixadores, assim como o mecanismo *gateway*, que é responsável pela interoperabilidade entre linguagens e conversão de tipos de dados. Também é apresentada uma ferramenta de suporte à geração das interfaces de co-simulação, que são constituídas de dois embaixadores configuráveis e um *gateway* para cada federado, gerado a partir de *templates* pré-definidos.

Palavras-chave: simulação distribuída, *co-design*, co-simulação, co-simulação distribuída e heterogênea, reuso, interoperabilidade, propriedade intelectual, HLA, DCB, Java, JNI.

Generation of Distributed Co-Simulation Models to the DCB Architecture

ABSTRACT

The increase in the complexity of embedded systems, composed by hardware and software parts, associated with the market pressure that requires new products in always shorter times, has led developers to consider the possibility of building systems by integrating already existing and previously validated components. These components may be developed by different teams or by third-part developers and are often designed using different methodologies, languages and/or abstraction levels. This heterogeneity makes even more complex the process of integration and verification of components, which is normally performed by simulation.

This work specifies generic and extensible mechanisms that offer support to cooperation between heterogeneous components in a distributed simulation environment, without imposing proprietary standards to data formats and to the description of the behavior and interface of components. These mechanisms are based on the DCB (Distributed Co-Simulation Backbone) architecture, oriented towards distributed heterogeneous co-simulation and inspired by the concepts of federate (simulation component) and federation (aggregate of components) that are defined by HLA (High Level Architecture), an interoperability standard for distributed simulations.

To give support to heterogeneous and distributed co-simulation, this work describes mechanisms that are responsible for cooperation and distribution of tasks, called ambassadors, as well as a gateway mechanism, which is responsible for the interoperability between languages and data type conversion. A supporting tool is also introduced for the generation of co-simulation interfaces, which are composed by two configurable ambassadors and one gateway for each federate, generated from predefined templates

Keywords: distributed simulation, co-design, co-simulation, distributed and heterogeneous co-simulation, reuse, interoperability, intellectual property, HLA, DCB, Java, JNI.

1 INTRODUÇÃO

O crescente aumento na complexidade dos sistemas embarcados, cada vez mais integrados em uma única pastilha, tem levado projetistas em busca de técnicas e metodologias mais eficientes para o projeto e implementação desses sistemas. O reuso de componentes é uma das técnicas mais empregadas, pois permite a otimização do processo de projeto de sistemas eletrônicos, chamado de *co-design*, através da integração de componentes previamente desenvolvidos e validados. Dessa forma, o projetista pode concentrar o esforço de validação na interação entre os componentes que fazem parte do sistema, o que pode reduzir consideravelmente o prazo para finalização do projeto.

O projeto de sistemas embarcados é caracterizado pela integração de componentes de hardware e software, que podem ter sido construídos por diferentes equipes de projeto, utilizando as mais diversas metodologias, linguagens e níveis de abstração. A descrição desses componentes, originários de domínios de aplicação diferentes, exige a utilização de linguagens distintas que possuem capacidades de representação diferenciadas ou facilitam a construção de determinados componentes. Por isso, na maioria dos sistemas, os componentes de hardware são descritos em linguagens para descrição de hardware como VHDL [IEE2000], SystemC [SYS2002] e Verilog [ACC2003], enquanto que os componentes de software são descritos em linguagens como C/C++ ou Java.

A atual carência de padrões e mecanismos que ofereçam um formalismo comum para integrar e validar um conjunto de componentes heterogêneos torna a etapa de validação uma tarefa complexa e um dos principais gargalos do processo de *co-design*. A etapa de validação, normalmente obtida através de co-simulação, oferece ao projetista a possibilidade de avaliar a funcionalidade e a performance dos componentes desde as fases mais iniciais do projeto. Uma das principais dificuldades encontradas nessa etapa é a construção das interfaces de co-simulação que, devido à heterogeneidade dos componentes, exige a existência de mecanismos de interoperabilidade que ofereçam suporte para lidar de forma padronizada com os diferentes padrões de interface e formas de comunicação que os componentes possuem.

As usuais abordagens para interoperabilidade entre linguagens são baseadas em soluções restritas ou proprietárias. As soluções utilizadas por ferramentas comerciais como CoCentric System Studio [SYN2003b] e Seamless CVE [MEN2003] são restritas a uma determinada combinação de simuladores e/ou linguagens para descrição de hardware. Soluções mais genéricas como o MCI [HES99] possibilitam a construção automática de modelos de co-simulação multi-linguagem, baseado em uma barramento de co-simulação distribuída que permite a integração de modelos assíncronos. Entretanto, nenhuma dessas soluções oferece mecanismos para interoperabilidade entre linguagens como C/C++, Java e VHDL.

Uma das soluções existentes para o problema da interoperabilidade é o padrão HLA (*High Level Architecture*) [DAH98], que é voltado para simulações distribuídas e heterogêneas. A arquitetura HLA fornece uma interface padronizada para simulações

distribuídas, chamadas de federações, e oferece suporte para o desenvolvimento de modelos de simulação baseados em componentes chamados de federados. Na arquitetura HLA, a comunicação entre federados é realizada através dos serviços disponibilizados pela infraestrutura de *runtime* (RTI). As interfaces de comunicação entre federado e RTI são encapsuladas em dois objetos distintos, chamados de embaixadores, um para o federado e outro para o RTI.

Esse estudo foi direcionado para a especificação de mecanismos voltados para a arquitetura de co-simulação DCB (*Distributed Co-Simulation Backbone*), proposta inicialmente por Mello em [MEL2001]. Essa arquitetura, inspirada nos conceitos do padrão HLA, disponibiliza uma infraestrutura de suporte que oferece serviços de cooperação e comunicação, permitindo a integração de componentes (federados) em um ambiente de co-simulação distribuída e heterogênea (federação).

A arquitetura DCB é dividida em três principais camadas: a camada DCB, os embaixadores e o *gateway*. A camada DCB, conceitualmente semelhante ao RTI-HLA, é responsável por oferecer serviços de suporte à cooperação, comunicação e sincronização entre federados heterogêneos. Os embaixadores, em conjunto com o *gateway*, são as entidades responsáveis pelas interfaces de comunicação entre o federado e a federação. Os embaixadores executam tarefas relacionadas ao gerenciamento de dados e à sincronização, e são configuráveis de acordo com as características de cada federado. O *gateway* é a camada na qual está encapsulada a interface do federado e onde estão implementados os mecanismos de interoperabilidade.

No contexto da arquitetura DCB, a federação é construída a partir da integração dos federados às interfaces de co-simulação e da configuração do ambiente de co-simulação, de acordo com as informações definidas pelo modelo da federação que é gerado pelo projetista em uma ferramenta visual de modelagem (desenvolvida em outra dissertação de mestrado). Levando em consideração a complexidade da tarefa de integração, o trabalho especifica uma ferramenta de apoio que faz a geração, para cada um dos federados que compõem a federação, da configuração dos embaixadores e do *gateway* apropriado, construído a partir da configuração de *templates* pré-definidos, que atendem às necessidades de integração de determinados tipos de federados.

Os mecanismos especificados nesse trabalho são validados através de um estudo de caso, onde é utilizado o modelo GPSAlerta [LIS2002], desenvolvido para ser conectado a um dispositivo GPS (*Global Position System*), com o objetivo de monitorar a posição geográfica do veículo no qual está instalado e emitir alertas ao usuário sobre a proximidade de pontos geográficos previamente cadastrados. Os experimentos foram baseados na modelagem, configuração e execução de três modelos do GPSAlerta, descritos em diferentes níveis de abstração, que permitiram a validação da funcionalidade e da correção dos mecanismos especificados.

1.1 Objetivos

O objetivo geral do trabalho é a especificação de mecanismos flexíveis e genéricos para auxiliar nas tarefas de integração e validação de componentes heterogêneos em um ambiente de co-simulação. Esses mecanismos devem oferecer suporte à execução de simulações compostas por componentes locais ou remotos, com o objetivo de validar o comportamento desses componentes e/ou a interação entre eles nas diferentes etapas do projeto de um sistema.

Os mecanismos especificados devem auxiliar na tarefa de construção das interfaces de co-simulação, que é considerada uma das tarefas mais dispendiosas e com alta probabilidade de erros. A construção de interfaces de co-simulação voltadas para a integração de uma ampla gama de modelos ou simuladores exige a definição de técnicas de interoperabilidade entre linguagens e também entre níveis de abstração, sem que haja a necessidade de imposição de padrões proprietários para formatos de dados e comportamento de componentes participantes da simulação.

A especificação e construção de tais mecanismos têm o objetivo de reduzir a complexidade das tarefas que envolvem o reuso de componentes previamente desenvolvidos, de modo que esses componentes, com o mínimo de alterações, possam ser integrados em outras simulações ou aplicações em desenvolvimento. Outro objetivo, que está diretamente ligado à popularização da utilização de técnicas de reuso em ambientes de *co-design* / co-simulação, é a definição de mecanismos que ofereçam aos desenvolvedores meios para disponibilizar seus componentes na Internet, permitindo que projetistas testem suas funcionalidades através da integração virtual e posterior simulação.

1.2 Contribuições

O trabalho contribui com a especificação de mecanismos alternativos para suporte à integração e cooperação entre componentes heterogêneos em um ambiente de co-simulação distribuída. Esses mecanismos são voltados para a infraestrutura de co-simulação DCB e foram especificados de forma modular, tornando possível futuras extensões como a inclusão de mecanismos de sincronização e gerenciamento de dados.

Um dos resultados obtidos a partir desse estudo foi a implementação de mecanismos flexíveis e genéricos que oferecem suporte à cooperação entre federados heterogêneos, podendo estes estar localizados em uma mesma máquina ou geograficamente distantes. Essa funcionalidade é possível graças ao modelo do barramento de comunicação, que permite a troca de mensagens, de forma transparente, entre federados locais e remotos. A possibilidade de instanciar federados remotos no ambiente de co-simulação dá ao projetista a opção de validar a funcionalidade e a integração desses componentes em seus sistemas, antes mesmo de adquiri-los. Além disso, permite a preservação da propriedade intelectual, pois os federados são simulados remotamente (no *site* do desenvolvedor) e, dessa forma, apenas eventos de simulação são trocados entre eles.

Outro ponto que pode ser destacado é a construção de uma ferramenta para dar apoio à geração e configuração das interfaces de co-simulação (*gateway* e embaixadores), possibilitando a redução da complexidade e do tempo consumido pela tarefa de integração dos componentes ao sistema nas diversas etapas do processo de *co-design*. A ferramenta gera o *gateway* de forma automática / semi-automática com base em *templates* pré-definidos voltados para um determinado conjunto de federados. Além disso, gera a configuração dos embaixadores, de acordo com as características do federado a quem representam, evitando tarefas de recompilação de código. A especificação da ferramenta permite futuras extensões, possibilitando a agregação de novos *templates* (voltados para a integração de diferentes componentes ou simuladores) e módulos para configuração dos mesmos.

1.3 Estrutura do Texto

O texto aborda temas que abrangem simulação distribuída, *co-design* e co-simulação, interoperabilidade entre simulações, arquiteturas e ambientes para simulação distribuída. O Capítulo 2 apresenta os conceitos envolvidos na área de simulação distribuída e o atual padrão para interoperabilidade entre simulações, o HLA (*High Level Architecture*). O Capítulo 3 abrange a área de *co-design* e co-simulação, assim como os principais trabalhos relacionados a essas áreas. O Capítulo 4 apresenta a especificação detalhada da arquitetura DCB (*Distributed Co-Simulation Backbone*) que é inspirada no padrão HLA, mas especificamente voltada para co-simulação distribuída e heterogênea. O Capítulo 5 descreve a implementação de um protótipo da arquitetura DCB e também apresenta mecanismos desenvolvidos para auxiliar nas tarefas que envolvem a construção de interfaces de co-simulação. O Capítulo 6 apresenta um estudo de caso, composto por três experimentos distintos, no qual são executadas co-simulações distribuídas utilizando componentes já existentes, com o objetivo de demonstrar as capacidades dos protótipos desenvolvidos. Por fim, no Capítulo 7 são apresentadas as considerações finais.

2 SIMULAÇÃO DISTRIBUÍDA

Os esforços da comunidade de simulação distribuída têm sido direcionados para o desenvolvimento de padrões e arquiteturas que ofereçam ganhos de desempenho e facilitem a interoperabilidade e o reuso de simulações.

Atualmente, a área de simulação distribuída está dividida entre duas grandes comunidades, atuando em linhas de pesquisa distintas: a Simulação Distribuída e Paralela (PADS - *Parallel and Distributed Simulation*) e a Simulação Interativa Distribuída (DIS – *Distributed Interactive Simulation*) [FUJ99]. Apesar dessas duas linhas de pesquisa possuírem objetivos diferentes, elas adotam alguns conceitos e metodologias comuns.

2.1 Simulação Paralela e Distribuída (PADS)

As pesquisas na área de simulação paralela e distribuída (*Parallel and Distributed Simulation* - PADS) estão voltadas para assuntos que envolvem a execução distribuída de simulações discretas baseadas em eventos [FUJ2001]. Nos últimos anos, os pesquisadores da área têm se concentrado principalmente em questões relacionadas à otimização de desempenho das simulações.

Um dos temas que mais motiva pesquisas na área são os mecanismos de sincronização, que têm o objetivo de garantir que os relacionamentos existentes no sistema a ser simulado sejam corretamente reproduzidos na simulação. Esses mecanismos normalmente assumem que uma simulação é composta por uma coleção de processos lógicos (LPs - *Logical Processes*) que se comunicam através da troca de mensagens ou eventos que possuem um *timestamp* associado.

O objetivo do mecanismo de sincronização é assegurar que cada LP processe os eventos de forma ordenada, de acordo com o *timestamp*. Essa exigência é chamada de LCC (*Local Causality Constraint*). Assim, se cada LP aderir ao LCC, a execução de uma simulação em um computador paralelo produzirá exatamente os mesmos resultados que a execução em um computador seqüencial [FUJ99].

Os algoritmos de sincronização podem ser divididos entre conservativos e otimistas, conforme detalhado a seguir.

2.1.1 Sincronização Conservativa

Os primeiros algoritmos baseados nessa abordagem foram desenvolvidos por Chandy e Misra [CHA79] e Bryant [BRY77], por isso, também são conhecidos como CMB (Chandy-Misra-Bryant).

Os algoritmos conservativos têm como principal objetivo prevenir a violação da LCC (*Local Causality Constraint*). Assim, a principal tarefa de qualquer protocolo conservativo é determinar quando é seguro processar um evento, ou seja, quando se pode garantir que nenhum evento contendo um *timestamp* menor será enviado posteriormente a um LP por outros LPs.

Diversos métodos baseados na abordagem conservativa foram desenvolvidos. A idéia por trás da maioria desses métodos é aumentar o *lookahead* disponível. O *lookahead* se refere à habilidade de prever o que acontecerá no futuro, baseado no conhecimento da aplicação e nos eventos que já foram processados. Assim, quanto maior o *lookahead*, maior será o número de eventos seguros que podem ser processados. Um dos principais problemas relacionados ao *lookahead* é o pessimismo ou o otimismo exagerado na determinação dos valores [VAC99].

Outro grande desafio imposto pelo método conservativo é evitar e recuperar situações de bloqueio (*deadlock*), que podem ocorrer quando um LP não possui eventos seguros para processar, causando, em muitos casos, o bloqueio da simulação. Um dos algoritmos mais conhecidos para evitar bloqueios utiliza o mecanismo de mensagens nulas (*null messages*) [FER95], [FUJ90], [NIC96].

2.1.2 Sincronização Otimista

Ao contrário da abordagem conservativa, os algoritmos otimistas permitem que violações de LCC ocorram. No entanto, estes devem estar aptos a detectá-las e recuperá-las, o que pode exigir um maior *overhead* computacional [FUJ2001].

A abordagem otimista oferece duas principais vantagens: (1) permite a exploração de um alto grau de paralelismo, ou seja, permite que eventos, em determinados casos, que seriam processados de forma seqüencial na abordagem conservativa, sejam processados de forma concorrente e (2) não necessitam de informações específicas da aplicação para executar de forma correta, ao contrário da abordagem conservativa, onde são necessárias informações para determinar quais eventos são seguros para serem processados.

Um dos mais conhecidos algoritmos otimistas é o *Time Warp*, que permite a operação assíncrona entre LPs que se comunicam através de mensagens com *timestamp* [THO98]. Esse mecanismo é baseado no paradigma do tempo virtual (*Virtual Time*) [JEF85], que é sinônimo de tempo de simulação.

Na abordagem otimista, cada LP mantém um relógio local, chamado de LVT (*Local Virtual Time*). O processo de incremento do LVT é baseado na observação do *timestamp* dos eventos da fila de entrada. O processo consiste em selecionar da fila de entrada o evento com o menor *timestamp*, processar o evento, e avançar o LVT para o *timestamp* desse evento processado.

Nos algoritmos otimistas, uma violação a LCC é detectada quando é recebido um evento com o *timestamp* menor que o LVT (*timestamp* do último evento processado).

Nesse caso, é necessário restaurar o LP ao seu estado correto, através da inicialização de um mecanismo de *rollback* e somente depois disso processar o evento recebido.

Os mecanismos de *rollback* são utilizados para resolver dois problemas: recuperar o dano causado pela execução de eventos fora de ordem (restaurar o estado do LP) e desfazer os efeitos das mensagens que o LP enviou por efeito do processamento prematuro de eventos da fila de entrada.

Um dos mecanismos mais conhecidos para desfazer os efeitos de mensagens enviadas de forma prematura é chamado de anti-mensagem [FUJ90]. Nesse mecanismo, para desfazer os efeitos de uma mensagem, basta que o LP envie a anti-mensagem correspondente à mensagem enviada previamente. Se essa mensagem já tiver sido processada, o receptor também precisa iniciar um processo de *rollback*, o que provavelmente resultará no envio de mais anti-mensagens. Assim, uma operação de *rollback* possivelmente resulta em um grande número de mensagens circulando no sistema para recuperar eventos processados de forma otimista. Isso pode tornar o processo de *rollback* o maior gargalo do sistema, limitando o aumento de desempenho.

Já para restaurar o estado do LP, uma das soluções mais utilizadas é o armazenamento do histórico de estados do LP no decorrer da execução (*checkpoints* podem ser utilizados para esse propósito). A utilização dessa abordagem gera dois problemas que precisam ser tratados: a impossibilidade de realizar *rollback* em determinadas situações (ex. operações de E/S) e o consumo contínuo de recursos de memória devido ao histórico que precisa ser mantido [FUJ2001], para o caso de algum *rollback* ocorrer.

Esses dois problemas normalmente podem ser solucionados através da utilização do conceito de GVT (*Global Virtual Time*). O GVT representa o menor limite de *timestamp* para qualquer futuro *rollback* [FUJ2001]. Ele é computado como o mínimo entre os relógios locais (LVTs) de todos os LPs e o *timestamp* de todas as mensagens em trânsito (mensagens enviadas mas ainda não recebidas) [NIC94].

Para contornar o problema do consumo excessivo de memória (decorrente do armazenamento do histórico) é comum a utilização de algoritmos que sejam capazes de recuperar a memória consumida por registros que definitivamente não serão mais utilizados. Essa alternativa, chamada de *fossil colection* [FER95], é baseada na busca e eliminação de eventos em que o valor do *timestamp* é menor que o valor do GVT.

2.2 Simulação Distribuída Interativa (DIS)

Enquanto que as pesquisas na área de PADS são amplamente voltadas para melhorar o desempenho das simulações, os pesquisadores da área de Simulação Distribuída Interativa (*Distributed and Interactive Simulation* - DIS) concentram-se na interoperabilidade e no reuso principalmente de simulações que envolvem interação com participantes reais e tarefas de tempo-real.

A área de DIS originou-se dos esforços do *US Department of Defense* (DoD) para desenvolver tecnologias que interconectassem simuladores operados por participantes reais (*human-in-the-loop*) [PAG99]. Para este propósito foi desenvolvida a tecnologia SIMNET (*SIMulator NETworking*), que demonstrou a viabilidade de utilizar simulações distribuídas para criar mundos virtuais (*virtual worlds*) para treinamento de soldados. Isso levou ao desenvolvimento de um conjunto de padrões para interconectar

simuladores, conhecido como protocolo DIS (*Distributed and Interactive Simulation*) [FUJ2001], que foi amplamente aceito nas comunidades militares e industriais, tornando-se um padrão para simulações distribuídas que envolviam tarefas de tempo-real.

Uma simulação DIS pode incluir: (1) participantes reais manipulando elementos virtuais, como tanques e simuladores de vôo, (2) elementos computacionais como simulações de jogos de guerra e (3) elementos reais como tanques e aeronaves.

Segundo [JEN97], o protocolo DIS é baseado nos seguintes princípios:

(a) Arquitetura objeto/evento: todas as entidades dinâmicas no ambiente informam todas as outras entidades sobre seu status e ações através da transmissão de pacotes de informação padronizados, chamados de PDUs (*Protocol Data Units*);

(b) Autonomia dos nodos de simulação: todos os eventos gerados por um simulador individual são enviados por *broadcast* e disponibilizados para todos os outros simuladores. O simulador receptor é responsável por determinar os efeitos causados por esse evento;

(c) Somente transmissão de mudança de estado: os simuladores transmitem somente informações sobre mudança de estado de entidades que ele simula. Atividades contínuas são transmitidas em uma taxa reduzida de atualização. Cada simulador, localmente, extrapola o último estado reportado pelos outros simuladores até que a próxima atualização de estado seja recebida. Isso é chamado de *dead-reckoning*.

Os princípios acima descritos permitem que simuladores minimizem o tráfego da rede, fazendo melhor uso do poder de processamento local.

Simulações DIS normalmente processam atualizações de estado na ordem em que elas são recebidas, pois a infraestrutura DIS não prevê serviços para ordenação de mensagens. Assim, diferentes simulações podem receber o mesmo conjunto de atualizações em diferentes ordens. Além disso, simulações DIS normalmente utilizam serviços de transmissão de mensagens não-confiáveis, sacrificando a confiabilidade em favor do desempenho.

Na década de 90, os conceitos de interoperabilidade e reuso do SIMNET foram aplicados em simulações de jogos de guerra, o que resultou no desenvolvimento do protocolo ALSP (*Aggregate Level Simulation Protocol*). Posteriormente, o *US Defense Modelling and Simulation Office* (DMSO), iniciou os estudos para estabelecimento de um novo padrão para modelagem e simulação com o objetivo de unificar e estender os padrões DIS e ALSP. Esse novo padrão foi chamado de HLA (*High Level Architecture*).

2.3 HLA (*High Level Architecture*)

HLA (*High Level Architecture*) é um padrão para interoperabilidade entre simulações. Foi originalmente desenvolvido pelo *US Department of Defense* para ser

um padrão para interoperabilidade entre simulações militares e recentemente foi aceita como padrão IEEE para simulação distribuída [STR2000].

O padrão HLA não define uma implementação específica, nem a utilização de qualquer software ou linguagem de programação particular [DAH98]. É um padrão voltado somente para a especificação de como simulações trabalham em conjunto para formar uma federação.

A arquitetura HLA fornece uma interface padronizada para simulações distribuídas e oferece suporte para o desenvolvimento de simulações baseadas em componentes, chamados de federados.

O principal objetivo do HLA é facilitar a interoperabilidade entre simulações heterogêneas e promover o reuso de simulações e de seus componentes [NAN99] [JEN97]. Dessa forma, sistemas de simulação já existentes podem ser combinados para formar novos sistemas de diferentes propósitos.

Os conceitos chave de HLA são as noções de federado e federação. Um federado pode ser definido como um participante de uma simulação distribuída baseada no padrão HLA [STR2000], podendo este ser uma simulação, um utilitário de suporte (ex. um aplicativo de visualização ou coletor de dados), um sistema de informação ou uma interface para um participante real (*live player*) [DAH98]. Já uma federação pode ser definida como uma aliança entre um ou mais federados atuando em conjunto dentro de uma simulação distribuída para alcançar um objetivo comum.

No padrão HLA, entidades do mundo real são modeladas como objetos, no entanto HLA não supõe a utilização de linguagens orientadas a objetos. Cada objeto possui (1) uma identificação que o distingue de outros objetos, (2) estado para o objeto e (3) uma descrição de comportamento que especifica como um objeto reage a mudanças de estados [FUJ96].

O relacionamento entre os objetos HLA é especificado através de (1) atributos que indicam aquelas variáveis de estado e parâmetros de um objeto que são acessíveis a outros objetos, (2) associação entre objetos (um objeto que é parte de outro) e (3) interações entre objetos que indicam a influência de um estado do objeto no estado de outro objeto [FUJ96].

Cada atributo de objeto HLA possui um proprietário (*owner*) que é responsável pela geração de atualizações para o valor de cada atributo. Inicialmente, a posse do atributo é do federado que instanciou o objeto [BRA99]. A principal restrição relacionada a um atributo de objeto é que, em qualquer instante, no máximo um federado poderá ter a posse de um determinado atributo, no entanto essa posse não é fixa e pode mudar durante a execução da simulação.

Com o objetivo de reduzir o tráfego da rede e limitar a quantidade de processamento de cada federado, o padrão HLA oferece mecanismos de publicação (*publish*) e subscrição (*subscribe*), que permitem ao federado atualizar e receber atualizações em atributos de objetos [DAH97]. Na inicialização, cada federado registra as classes de objetos e os atributos associados que ele irá representar (*publish*). Além disso, o federado também registra classes de objetos, atributos e interações que ele necessita para executar suas tarefas (*subscribe*). Essas publicações e subscrições são dinâmicas e podem ser modificados durante a execução [BRA99].

Para facilitar a interoperabilidade e reusabilidade, HLA faz a separação entre a funcionalidade de simulação oferecida pelos membros da simulação distribuída e o conjunto de serviços básicos para troca de dados, comunicação e sincronização.

Funcionalmente, uma federação HLA pode ser particionada em três componentes (veja FIGURA 2.1). O primeiro componente são as simulações, ou mais especificamente os federados.

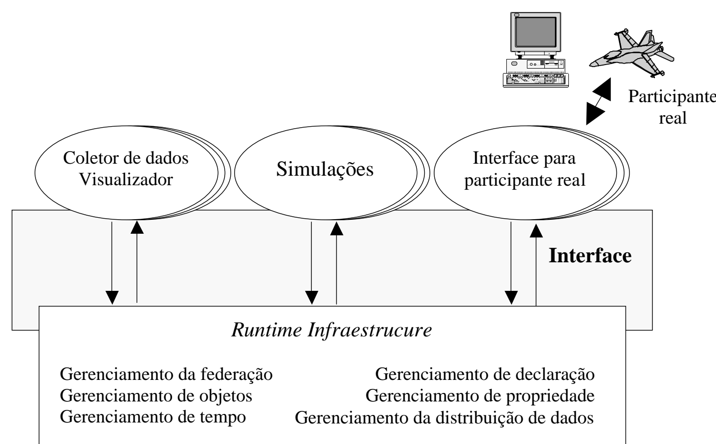


Figura 2.1: Visão funcional de uma federação HLA (Fonte: [DAH98])

O segundo componente funcional é o RTI (*Runtime Infraestructure*), que é considerado o sistema operacional distribuído da federação. O RTI oferece um conjunto de serviços de propósito geral que dão suporte a interações federado-para-federado, gerenciamento da federação e funções de suporte [DAH98]. Todas as interações entre federados obrigatoriamente ocorrem através do RTI.

O terceiro componente é a interface com o RTI. A HLA *Runtime Interface Specification* oferece uma forma padronizada para os federados interagirem com o RTI, para requisitar serviços e para responder a requisições recebidas do RTI. Essa interface é independente da implementação dos modelos específicos de objetos e dos requisitos para troca de dados de qualquer federação.

Formalmente, HLA é definida por três componentes principais: a especificação de interface (*HLA Interface Specification*), o HLA *Object Model Template* (OMT) e as regras HLA (HLA *Rules*), conforme detalhado a seguir.

2.3.1 Especificação de interface HLA (HLA *Interface Specification*)

A especificação de interface HLA descreve os serviços que o federado pode usar para comunicar-se com outros federados via *Runtime Infraestructure* (RTI). A especificação da interface também descreve quais serviços podem ser utilizados por um federado e quais serviços não são fornecidos.

Existem seis classes de serviços:

1. *Federation management* – oferecem funções básicas necessárias para criar e operar uma federação;
2. *Declaration management* – suportam gerenciamento eficiente de troca de dados, através de informações fornecidas pelos federados definindo os dados que estes irão fornecer e solicitar durante a execução da federação;
3. *Object management* – oferece serviços para criação, exclusão, identificação e outros serviços no nível de objeto;
4. *Ownership management* – serviços de suporte para transferência dinâmica de propriedade de objetos/atributos durante a execução;
5. *Time management* – dão suporte à sincronização nas trocas de dados da simulação;
6. *Data distribution management* – dão suporte ao roteamento eficiente de dados entre os federados durante a execução da federação.

A interface HLA é bi-direcional e está encapsulada dentro do paradigma de embaixador (veja FIGURA 2.2). Um federado comunica-se com o RTI utilizando seu embaixador RTI. Da mesma forma, o RTI comunica-se com o federado via embaixador federado. Na visão do programador do federado, esses embaixadores são objetos e a comunicação entre os participantes é executada através da chamadas de métodos desses objetos. Assim, os serviços definidos na especificação da interface são métodos do embaixador RTI ou do embaixador federado [STR2000].

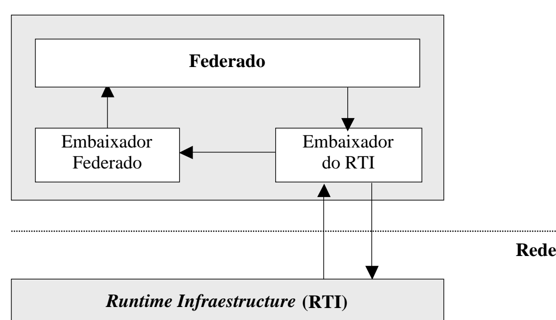


Figura 2.2: O paradigma dos embaixadores na interface HLA (Fonte: [STR2000])

2.3.2 Modelos de Objetos HLA (HLA *Object Models*)

Modelos de objetos HLA são a descrição dos elementos essenciais compartilhados pela simulação ou federação em termos de "objetos" [DAH98]. HLA especifica dois tipos de modelos de objetos: o HLA *Federation Object Model* (FOM) e o HLA *Simulation Object Model* (SOM).

O HLA FOM descreve o conjunto de objetos, atributos e interações, os quais são compartilhados através da federação. Já o HLA SOM descreve a simulação (federado) em termos de tipos de objetos, atributos e interações que ele pode oferecer para futuras federações. O SOM não tem ligação com as informações internas de projeto da simulação. Ele somente fornece informações sobre a capacidade da simulação de trocar

informações como parte de uma federação. A disponibilidade do SOM facilita a avaliação do quanto o federado é apropriado para participação em uma determinada simulação.

O padrão HLA não coloca obstáculos no conteúdo dos modelos de objetos, entretanto exige que uma abordagem padronizada para documentação seja utilizada, a HLA *Object Model Template* (OMT), que é um *template* geral para especificação das tabelas que precisam ser documentadas [STR2000]. Os *templates* são os meios para compartilhamento aberto de informações através da comunidade. Para facilitar o reuso, esses *templates* completos poderiam ser disponibilizados para que ferramentas automatizadas pudessem realizar buscas sobre os dados do modelo deobjetos.

2.3.3 Regras HLA (HLA *Rules*)

As regras HLA resumem os três princípios chave por trás do padrão HLA. As regras são divididas em dois grupos: regras de federação e de federado. A seguir são mostradas, de forma resumida, as 10 regras que resumem o padrão HLA, sendo cinco delas para federações e as outras cinco para federados:

Regras para federações [USD98]:

1. Federações deverão ter um FOM (*Federation Object Model*), documentado de acordo com o OMT (*Object Model Template*);
2. Em uma federação, todas as representações de instâncias de objetos relacionadas à simulação devem estar no federado e não no RTI;
3. Durante a execução da simulação, toda a troca de dados entre federados deve ocorrer através do RTI;
4. Durante a execução da federação, os federados devem interagir com o RTI de acordo com a especificação de interface HLA;
5. Durante a execução da federação, um atributo de uma instância deve ser propriedade de no máximo um federado num dado instante de tempo.

Regras para federados [USD98]:

6. Federados deverão ter um SOM (*Simulation Object Model*), documentado de acordo com o OMT (*Object Model Template*);
7. Federados deverão estar aptos a atualizar e/ou refletir quaisquer atributos e enviar e/ou receber interações, como especificado em seus SOMs;
8. Federados deverão estar aptos a transferir e/ou aceitar propriedade sobre atributos dinamicamente durante a execução da federação, como especificado em seus SOMs;
9. Federados deverão estar aptos a variar as condições de atualização dos seus atributos, como especificado nos seus SOMs;

10. Federados devem estar aptos a gerenciar o tempo local, de forma a permitir a coordenação da troca de dados com outros membros da federação.

2.3.4 Abordagens para integração de federados através da interface HLA

A interface HLA é a camada de software entre o sistema de simulação e o RTI. Possui a tarefa de traduzir e gerenciar o fluxo de informação e de controle entre o RTI e o federado, e assim prover funcionalidade HLA para o sistema de simulação.

É altamente desejável para os desenvolvedores de simulações que uma interface HLA fosse desenvolvida uma só vez para uma variedade de modelos. Entretanto, em muitos casos são necessárias interfaces dependentes do modelo de simulação.

Para permitir a interação entre o RTI e o HLA, é necessário que a ferramenta de simulação acesse a API (*Application Programming Interface*) HLA, e para isso é inevitável programação de baixo nível em uma linguagem de programação tradicional [STR2000].

Atualmente, APIs escritas em diferentes linguagens (CORBA IDL, C++, Ada e Java) estão incorporadas à especificação de interface. Para federados desenvolvidos em outras linguagens devem ser utilizados meios alternativos para construir as chamadas ao embaixador do RTI.

Além de chamar métodos do embaixador RTI, os federados também devem fornecer seu próprio objeto embaixador que contém métodos, chamados de *callback functions*, que podem ser invocados pelo RTI. Esses métodos devem também obedecer ao padrão HLA como definido na especificação de interface (HLA *Interface Specification*).

Interfaces HLA para sistemas de simulação podem ser classificadas com relação a duas perspectivas distintas: a perspectiva do programador (a pessoa que está desenvolvendo a interface HLA e programando em baixo-nível) e a perspectiva do usuário (a pessoa desenvolvendo o modelo baseado em HLA) [STR2000].

De acordo com a perspectiva do programador que está desenvolvendo a interface HLA, foram levantadas por [STR2000] quatro estratégias para acessar a API HLA:

1. Re-implementação da ferramenta – É a solução mais direta se o código do sistema de simulação é disponível.
2. Extensão de código intermediário – Levando em conta o fato de que algumas ferramentas de simulação traduzem o modelo criado na ferramenta de modelagem em uma linguagem de programação, como C e C++, pode-se modificar esse código intermediário adicionando extensões HLA. Uma solução automatizada é desejável.
3. Uso de uma interface de programação externa - Essa solução é satisfatória para ferramentas que oferecem uma arquitetura aberta e extensível. A ferramenta poderia oferecer uma "*interface library*", como uma DLL no Windows, com a habilidade para chamar funções ou métodos dessas bibliotecas.
4. Acoplamento através de um programa *gateway* – Essa é outra solução para ferramentas que não podem acessar a API HLA. É desenvolvido um programa *gateway* que se comunica com a ferramenta de simulação através

de meios apropriados (ex: filas, *pipes*, portas e rede) dependendo das capacidades da ferramenta de simulação.

Já na perspectiva do usuário, que está construindo um modelo de simulação e deseja utilizar serviços HLA, existem duas abordagens para adicionar funcionalidades HLA ao modelo de simulação: de forma explícita ou implícita.

Na abordagem explícita o desenvolvedor do modelo deve explicitar o uso da funcionalidade HLA, por exemplo, chamando funções do RTI. Para isso, é necessário que a ferramenta de simulação ofereça formas de realizar o mapeamento entre a API HLA e a API específica da ferramenta que o usuário está utilizando. Assim, o usuário pode definir funções para serem chamadas de dentro de seus modelos. Essas funções corresponderiam aos métodos definidos na especificação de interface (HLA *Interface Specification*). Levando em conta que a maioria das ferramentas de simulação não oferece funções *callback*, formas alternativas devem ser usadas para transferir dados de volta para a ferramenta. Isso é necessário porque o paradigma de programação HLA exige a implementação de funções *callback* as quais recebem dados de entrada através de chamadas feitas pelo RTI.

Na abordagem implícita, toda a funcionalidade HLA fica oculta para o desenvolvedor do modelo. O sistema de simulação manipula toda a comunicação HLA internamente e o desenvolvedor da simulação não tem consciência disso. As chamadas ao RTI são executadas implicitamente quando for apropriado (ex. quando um atributo de um objeto é modificado). Entretanto, a solução implícita exige que algumas tarefas sejam executadas automaticamente, como: conversão de tipos de dados específicos da ferramenta, sincronização com outros federados (utilizando mecanismos padronizados oferecidos pelo HLA), geração de atualizações/interações quando as variáveis do modelo forem alteradas. A abordagem implícita somente é aplicável se o código fonte do sistema de simulação estiver disponível, o que pode ser inviável em muitos casos.

2.3.5 Interoperabilidade no padrão HLA

A interoperabilidade foi um dos fatores motivadores do padrão HLA. No entanto, mesmo com todas as vantagens oferecidas pela arquitetura HLA, ainda não é possível (devido à tecnologia atual) alcançar a tão desejada "interoperabilidade universal", onde qualquer simulação poderia interoperar com qualquer outra simulação, pois é necessário que os modelos "falem a mesma linguagem".

A necessidade das simulações cooperantes conhecerem detalhes da modelagem de objetos e da comunicação entre eles torna a interoperabilidade mais fácil entre componentes desenvolvidos para um mesmo ambiente de simulação (homogêneos) e mais complexa entre componentes baseados em arquiteturas e sistemas de comunicação diferentes (heterogêneos).

Os principais desafios relacionados à interoperabilidade, encontrados no desenvolvimento do padrão HLA, podem ser classificados em três principais assuntos: (1) formato padronizado para troca de dados (DIF - *Data Interchange Format*), (2) diferentes mecanismos de gerenciamento de tempo e (3) a uniformidade e consistência na representação dos modelos do federado. Esses desafios são descritos de forma detalhada em [NAN99].

Ainda que existam outras tecnologias que forneçam interoperabilidade em nível de aplicação em geral (ex. CORBA) e serviços de interoperabilidade na área de simulação, não existe tecnologia, exceto HLA, que declare ser aplicável em uma ampla gama de simulações, sem que seja limitada a um nicho específico de mercado [STR2000]. Isso ocorre porque as soluções atuais para simulação distribuída são proprietárias e os simuladores que estão interconectados não podem conversar com simuladores baseados em outras soluções proprietárias. Adicionalmente, essas soluções muitas vezes adotam "atalhos" para resolver os problemas de padronização de troca de dados e sincronização dos relógios de simulação. Levando em conta todos esses fatores, atualmente a arquitetura HLA pode ser considerada o estado da arte na área de simulação distribuída.

3 CO-DESIGN E CO-SIMULAÇÃO

O aumento da complexidade no projeto de sistemas compostos por partes de hardware e software (*co-design*) tem levado desenvolvedores a considerar a integração de componentes reutilizáveis em seus projetos. O reuso de componentes previamente testados diminui de forma significante a complexidade da tarefa de verificação do sistema [PAN2000], pois permite que o desenvolvedor concentre-se principalmente nas tarefas de integração dos componentes e validação do sistema como um todo.

Os componentes, também conhecidos como IPs (*Intellectual Property Blocks*) ou *Cores*, normalmente são desenvolvidos por diferentes equipes de projeto ou por terceiros e muitas vezes são projetados utilizando diferentes metodologias, linguagens e/ou níveis de abstração. Devido a essa heterogeneidade, a etapa de verificação do sistema (nesse caso obtida através da co-simulação) torna-se uma tarefa complexa e que consome muito tempo e, por isso, pode ser considerada o gargalo mais crítico do processo de *co-design*.

3.1 Co-design

O processo de *co-design*, ilustrado de forma simplificada na FIGURA 3.1, é uma metodologia para projeto concorrente de hardware/software utilizada no desenvolvimento da maioria dos sistemas eletrônicos modernos [COU95].

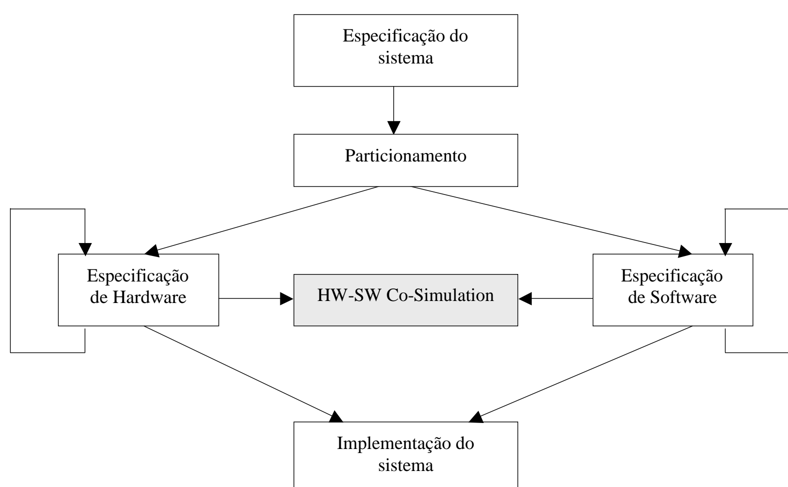


Figura 3.1: Hardware / Software *co-design* (Fonte: [HUB98])

Na maioria dos processos de *co-design*, o componente de software é um programa escrito em C ou C++ e o componente de hardware é escrito em linguagens de descrição de hardware como VHDL e Verilog. Descrições detalhadas do processo de *co-design* e as mais conhecidas metodologias utilizadas podem ser encontradas em [BUC94], [CHI94], [WOL94] e [ERN98].

De uma forma geral, em uma metodologia de *co-design*, o principal objetivo é desenvolver um sistema com determinados requisitos de performance, consumo de potência e área e ao mesmo tempo com o menor custo possível. Dependendo da aplicação, podem ser necessárias diferentes arquiteturas utilizando diferentes combinações de componentes de hardware e software [VER96]. Assim, os melhores sistemas mistos surgem através do reconhecimento de quais partes do sistema são melhor executadas em software e quais são melhor implementadas em hardware.

O processo de *co-design* é iniciado com a especificação do sistema como um todo, em uma linguagem natural. Essa especificação é refinada através da análise de requisitos e finalmente traduzida para uma linguagem que permita a verificação funcional. Nesse estágio, uma primeira simulação do sistema é executada com o objetivo de avaliar a funcionalidade em um alto nível de abstração. Essa verificação funcional deve ser realizada a partir dos estágios iniciais do projeto, para tornar mais fácil o desenvolvimento do sistema, diminuir o custo e o tempo de projeto.

Após isso, é feito o particionamento entre hardware e software, levando em consideração fatores como performance, flexibilidade e custo [FOR98]. Depois da divisão do sistema, a co-simulação pode ser utilizada para obter informações mais detalhadas sobre o comportamento desse sistema.

A etapa final do processo de *co-design* é iniciada depois que todas as simulações foram executadas com sucesso. Nessa etapa o software pode ser compilado para código de máquina e o hardware implementado em, por exemplo, um FPGA ou um ASIC. Assim, finalmente podem ser realizados testes para validar a funcionalidade do sistema real.

3.1.1 Linguagens de Especificação e Projeto

Nas várias etapas do projeto de sistemas embarcados é necessário descrever o comportamento dos componentes e a comunicação entre eles em diferentes níveis de abstração. Os componentes descritos podem ser partes de software ou hardware de um sistema, o que exige um modelo de computação adequado para representar cada um desses diferentes domínios [EDW97]. Diversas linguagens de especificação e de projeto têm sido adotadas para tratamento dos diferentes modelos de computação e níveis de abstração, entre elas: C, C++, Java, VHDL, Verilog, SystemC e outras.

A especificação inicial de um sistema normalmente é feita em um alto nível de abstração, que pode ser chamada de modelo comportamental. Para fins de validação do comportamento do sistema é importante que, nesse modelo inicial, a linguagem utilizada seja executável. Para isso, atualmente têm sido utilizadas linguagens como C/C++, que são apropriadas para descrições de software e para especificações funcionais de alto nível. Entretanto, essas linguagens não possuem uma semântica adequada para a descrição de aspectos de hardware, o que exige a utilização de linguagens adicionais.

Para descrever componentes de hardware, as linguagens mais utilizadas são VHDL e Verilog. A utilização dessas linguagens é interessante, pois permitem que as descrições dos componentes sejam utilizadas tanto para simulação quanto para síntese. VHDL, no entanto, é uma linguagem mais orientada para simulação, de tal modo que algumas de suas construções não são sintetizáveis, o que força ferramentas de síntese a aceitarem apenas um determinado subconjunto da linguagem e/ou estilo de descrição.

Com o objetivo de oferecer uma única linguagem / formalização para atender aos diferentes domínios de aplicação (hardware / software) que envolvem o projeto de sistemas embarcados, foi introduzida a linguagem SystemC [SYS2002]. A plataforma SystemC é baseada na linguagem C++, que é estendida (através de bibliotecas de funções) para oferecer semânticas / construções apropriadas para descrição de hardware como portas, sinais, relógios e outros. A linguagem SystemC, em sua primeira versão, oferecia uma semântica de hardware muito próxima de VHDL, permitindo somente descrições de comunicação entre processos em baixos níveis de abstração. Na versão 2.0 já é possível a modelagem de mecanismos de comunicação mais abstratos, baseados em construções de alto nível como canais, interfaces e eventos.

Algumas tentativas para descrever e simular sistemas tem sido feitas com a linguagem Java [HEL97] [KUH99] [ITO2001], com o objetivo de facilitar a modelagem e o reuso de componentes. Essas abordagens aproveitam a vantagem da portabilidade que a linguagem Java oferece, permitindo que uma única descrição possa ser executada em diversas plataformas sem necessidade de qualquer adaptação. Contudo, a descrição de componentes em Java não oferece qualquer vantagem em comparação com C++, pois a dificuldade para modelar componentes em diferentes níveis de abstração pode ser considerada a mesma nas duas linguagens.

Nenhuma das linguagens citadas atende simultaneamente aos requisitos exigidos pelos múltiplos domínios e níveis de abstração necessários para a descrição de componentes de hardware e software. Além disso, soluções empregadas em ferramentas comerciais como CoCentric System Studio [SYN2003b], da Synopsys e Seamless CVE [MEN2003], da Mentor são restritas a uma determinada combinação de simuladores e/ou linguagens para descrição de hardware. Assim, torna-se necessária a especificação de mecanismos genéricos, que utilizem uma semântica comum e permitam a descrição e combinação de diferentes modelos de computação (cada um implementado em uma linguagem específica), atendendo às necessidades intrínsecas dos diferentes domínios de aplicação e níveis de abstração.

3.2 Co-Simulação

A co-simulação pode ser descrita como uma simulação composta por sistemas heterogêneos, onde componentes de hardware e software interagem. É considerada uma das etapas mais importantes do processo de *co-design*, pois permite que projetistas executem experimentos que seriam na prática muito difíceis de realizar com protótipos [HIN97].

A co-simulação permite que diversos modelos, especificados em diferentes níveis de abstração [WOL94], executem em conjunto utilizando um formalismo comum. Essa técnica auxilia nas seguintes tarefas:

1. Testes para seleção de componentes [SUN97];
2. Decisões relacionadas ao particionamento do sistema [COU95] [KIM95];
3. Validação da funcionalidade e performance dos componentes que estão trabalhando em conjunto no sistema;
4. Verificação da implementação final do sistema [KIM95], ou seja, se o resultado da integração está de acordo com a especificação descrita pelo projetista.

Um fator importante para a co-simulação é a escolha de um modelo de processador adequado, pois ele faz a conexão entre as partes de hardware e software do sistema. A escolha de um modelo de processador em um determinado momento do processo de *co-design* é crucial para alcançar o resultado desejado para a co-simulação.

Normalmente, os projetistas fazem a escolha do modelo do processador levando em consideração os seguintes aspectos: disponibilidade do modelo, desempenho (considerando o tempo de simulação), precisão de tempo, possibilidade de acesso aos estados internos (possibilidade de *debug*). Rowson [ROW94] descreve algumas das principais técnicas para co-simulação baseadas principalmente na disponibilidade do modelo do processador.

As técnicas dependentes do modelo de processador são classificadas de acordo com os diferentes níveis de abstração em que o processador foi modelado, o que é um fator determinante para definir a relação entre a precisão e o desempenho da co-simulação. Essas técnicas, descritas com mais detalhes em [ROW94], são:

(a) *Nano-Second Accurate* - é o modelo de software mais preciso, mas com o menor desempenho. Utiliza o modelo do processador que possui completa funcionalidade e precisão de tempo de nano-segundos para todos os pinos.

(b) *Cycle Accurate* - oferece as transições corretas a cada avanço de relógio, mas sem levar em consideração a temporização. Esse modelo é bem mais simples e oferece melhor desempenho que o *Nano-Second Accurate*, pois não precisa escalonar eventos em cada pino separadamente.

(c) *Instruction Set Accurate* – é o modelo que utiliza precisão de conjunto de instruções. Isso garante a emulação precisa do conjunto de instruções, o que significa que valores em registradores e memória são corretamente modelados. Essa técnica, entre as que exigem o modelo do processador, é a que possui melhor desempenho, embora perca em precisão.

Quando não está disponível o modelo funcional do processador, o software pode ser compilado e executado em uma máquina e a simulação pode ser igualmente executada. As técnicas que podem ser utilizadas nesse caso são:

(a) *Synchronized Handshake* – técnica utilizada quando o software e o hardware comunicam-se através de métodos assíncronos, de forma que o tempo de comunicação não produz efeitos na funcionalidade. Nessa técnica, o software é compilado e comunica-se com o simulador do hardware

através de uma sincronização do tipo *handshake* [HIN97a]. Nessa técnica a velocidade do sistema é limitada pelo desempenho do simulador do hardware.

(b) *Virtual Operating System* – nessa técnica, o hardware e o sistema operacional são abstratos e no lugar deles existe um sistema operacional virtual, o qual pode ser visto da mesma forma que o sistema real (na perspectiva do software). Essa técnica oferece um bom desempenho, mas não permite que o modelo do hardware seja depurado de nenhuma forma.

(c) *Bus Functional* – o modelo *bus functional* descreve somente a interação entre o processador e os componentes externos, não executando qualquer tipo de software. Porém, sua interface é configurada (programada) para comportar-se, aparentemente, como se o software estivesse rodando no processador. Esse modelo normalmente é utilizado para testar e depurar o lado do hardware da interface hardware / software.

Segundo [HIN97a], para validação de sistemas embarcados, muitas vezes não é necessária preocupação com detalhes internos de operação do processador. O trabalho é muito mais concentrado em questões relacionadas a tempo, funcionalidade de hardware e software e suas respectivas interfaces. Assim, qualquer modelo de processador que execute o código de forma correta e ofereça os eventos de interface na forma e no tempo correto será suficiente.

Um dos maiores problemas encontrados na co-simulação está relacionado com o desenvolvimento das interfaces entre hardware e software e a forma como estas irão se comunicar. Para diminuir a complexidade dessa tarefa normalmente são utilizados diferentes níveis de abstração no decorrer do processo de *co-design*. Isso permite que o sistema seja inicialmente simulado utilizando um modelo comportamental (*behavioral*) e, após sucessivos refinamentos (decorrentes do processo de *co-design*) chegar ao mais baixo nível (*register-transfer-level*), que representa a implementação final do sistema.

Simulações em alto nível de abstração normalmente são executadas nas fases iniciais do projeto, com o objetivo de validar a funcionalidade dos componentes e suas interações, permitindo a redução do tempo de projeto e a ocorrência de erros [CES2002]. Nessas simulações, a interface entre hardware e software é modelada em alto nível de abstração utilizando, por exemplo, primitivas *send* e *receive*. Esse tipo de simulação, que pode ser chamada de comportamental (*behavioral*), é geralmente composta pelo modelo comportamental do hardware e pelo protótipo simplificado do software. Essa abordagem, que não leva em conta aspectos de sincronização, permite a redução do tempo de validação [PAN2000], mas não é interessante para tarefas de avaliação de desempenho [ADA96] e verificação final do sistema.

Já simulações modeladas em um baixo nível de abstração são executadas no decorrer do processo de *co-design*, com o objetivo de identificar possíveis erros e estimar desempenho, fazendo com que a especificação do sistema possa ser revisada. Em níveis mais baixos de abstração, a interface entre hardware e software pode ser modelada em termos de pinos de uma CPU ou ligações a um barramento [ADA96]. Normalmente, no nível mais baixo nível de abstração, também chamado de RTL (*register-transfer-level*), o modelo do hardware já possui detalhes de funções lógicas e registradores e o protótipo do software está codificado em uma linguagem de alto nível. Nesse tipo de simulação a validação temporal do modelo é necessária [SUN97], o que

exige a sincronização entre os componentes de hardware e software. Os modelos de comunicação no nível RT (*register-transfer*) são considerados computacionalmente caros, devido ao grau de refinamento da interação entre os componentes.

Apesar da utilização de diferentes níveis de abstração oferecer vantagens para as diversas etapas do processo *co-design*, ainda existem dificuldades que precisam ser superadas. Ortega [ORT99] afirma que o nível de abstração das interfaces, oferecido pelas linguagens de descrição de hardware tradicionais, não é suficiente para lidar com o crescimento da complexidade no projeto de sistemas e circuitos.

Outro problema, levantado por Rowson [ROW97], está relacionado com os métodos utilizados para projetar sistemas, onde a comunicação é misturada com o comportamento do componente, de forma que é difícil tornar essa comunicação abstrata. Por isso, ele propõe a utilização de uma metodologia que permite a clara separação entre o comportamento e comunicação nos componentes. Essa metodologia, chamada de *Interface-based Design*, permite a utilização, de forma transparente, de diferentes níveis de abstração tanto no comportamento interno do componente quanto nas interfaces de comunicação.

Metodologias que oferecem a clara separação entre interface e comportamento facilitam que os componentes e suas interfaces sejam inicialmente modelados em altos níveis de abstração e sofram sucessivos refinamentos no decorrer do processo de *co-design*. A utilização dessas metodologias permite a redução do tempo de projeto, pois a troca de nível de abstração da interface pode ser feita sem a necessidade de alterar o código do componente.

Uma abordagem como a *Selective Focus* [HIN97], disponibilizada em um ambiente de simulação, permite ao projetista escolher dinamicamente o nível de detalhamento para cada uma das partes do sistema que está sendo simulado. Isso possibilita que partes mais detalhadas do sistema sejam executadas localmente, enquanto que outros elementos (já validados ou de menor importância para o sistema) podem estar localizados em máquinas remotas, o que reduziria de forma considerável a sobrecarga gerada pela comunicação entre os componentes.

Na maioria dos casos, o ideal é que a co-simulação possua tanto um alto grau de detalhamento quanto um bom desempenho [HIN97a]. No entanto, simulações detalhadas tendem a ser excessivamente lentas e muitas vezes inviáveis. Uma técnica que permita a utilização, ao mesmo tempo, de diferentes níveis de abstração (multi-nível) tanto para o comportamento quanto para a interface do componente permite a melhora no desempenho geral do sistema, pois torna possível que um ou mais módulos já validados sejam simulados com um menor número de detalhes, enquanto que os módulos que são realmente o foco da validação podem ser simulados em um alto grau de detalhamento.

3.2.1 Ambientes de co-simulação

Alguns dos principais problemas encontrados no processo de co-simulação estão diretamente relacionados com a heterogeneidade dos modelos. Na co-simulação, vários simuladores independentes, cada um otimizado para uma tarefa especial, são combinados em um ambiente comum [SCH95]. Por isso, é importante considerar os ambientes onde a co-simulação será executada, os quais podem ser diferenciados por três principais fatores [JER99]:

1. Máquina de simulação – a co-simulação pode ser executada utilizando uma única máquina de simulação (homogênea) ou utilizando duas ou mais máquinas de simulação diferentes (heterogênea):
   (a) Co-simulação em ambientes homogêneos – geralmente não oferece grandes problemas, pois as informações sobre os estados internos dos subsistemas (hardware e software) estão sempre acessíveis e podem ser utilizadas para coordenar as interações entre as entidades de simulação.
   (b) Co-simulação em ambientes heterogêneos – devido à utilização de diferentes simuladores, é essencial levar em conta aspectos relacionados à comunicação, sincronização e conversão de tipos de dados entre os simuladores [BIS97] [HUB98].

2. Modelo temporal – a co-simulação pode ser não-temporizada ou temporizada, conforme detalhado a seguir:
   (a) Não-temporizada – nesse tipo de co-simulação o tempo de execução das operações não é tratado. Essa abordagem permite somente validação funcional. O tempo necessário para a computação não é levado em conta durante a co-simulação. Assim, a comunicação entre os módulos pode ser feita através de um protocolo do tipo *handshake* onde a ordem dos eventos é utilizada para sincronizar os dados e as trocas de sinais de controle.
   (b) Temporizada – nessa co-simulação, cada participante tem seu relógio local e o tempo de execução da computação é levado em conta. O tempo pode ser definido em diferentes níveis de granularidade os quais definem diferentes níveis da co-simulação.

3. Modelo de sincronização – No caso da co-simulação heterogênea, utilizando duas ou mais máquinas de simulação, um fator importante para o ambiente de co-simulação é a definição do modelo de comunicação e sincronização entre os diferentes simuladores. Por isso, é essencial a existência de uma infraestrutura que ofereça interfaces padronizadas, mecanismos de sincronização e troca de dados.

Para tratar o problema da comunicação entre simuladores heterogêneos que precisam cooperar em uma mesma co-simulação, em primeiro lugar, é necessário que o ambiente de co-simulação entenda a semântica de todos os modelos envolvidos e como ações em um domínio podem afetar o estado de outro [ADA96]. A próxima tarefa é possibilitar a comunicação entre os simuladores e para isso é necessário oferecer interfaces adequadas para cada um dos envolvidos. Adicionalmente, para que a comunicação seja realizada de forma correta também é necessária a existência de mecanismos para traduzir as informações e executar a transferência de dados [BIS97] de uma forma consistente e unificada [KIM95].

Para tratar o problema da sincronização em ambientes heterogêneos de co-simulação é necessária a definição de um mecanismo de sincronização entre os componentes de

hardware e software, que coordene os relógios locais dos simuladores, o que é um problema bastante semelhante ao encontrado na área de simulação distribuída. Por isso, é natural que os algoritmos desenvolvidos na área de simulação distribuída sejam utilizados, talvez com algumas modificações, para solucionar o problema de sincronização na co-simulação [VAC99]. Como citado no Capítulo 2, a tarefa de sincronização pode ser executada através de duas principais abordagens: a otimista e a conservativa. Algumas das soluções propostas podem ser encontradas em [YOO97], [COU95], [BIS97].

3.3 Trabalhos relacionados

Atualmente existem diversos projetos em andamento, envolvendo diferentes linhas de pensamento dentro da comunidade de *co-design* e co-simulação. Dentre eles Ptolemy, IPChinook, DISCOE, S^3E^2S, MCI e ROSES.

3.3.1 Ptolemy

O projeto Ptolemy [LEE2001] é voltado para modelagem, simulação e projeto de sistemas heterogêneos, mais especificamente focado em sistemas embarcados (*embedded systems*). Atualmente, em sua segunda geração é chamado de Ptolemy II.

O Ptolemy II oferece uma infraestrutura unificada para dar suporte à construção e interoperabilidade de modelos executáveis (simulações) que são desenvolvidos sob uma ampla variedade de modelos de computação. Esses modelos de computação são implementados em diferentes domínios (*domains*) que tratam aspectos de concorrência e temporização de diferentes maneiras, fornecendo mecanismos de interação distintos aos componentes.

O Ptolemy II permite, por exemplo, que em projetos de hardware os componentes sejam modelados em nível de detalhamento diversificado, envolvendo desde o nível comportamental (*behavioral*) até o projeto do circuito.

Entre os principais avanços do Ptolemy II em relação ao seu antecessor estão: utilização de Java, projeto baseado em componentes, concorrência e integração com a rede. Ainda são previstas futuras extensões que incluem: utilização do padrão XML (*eXtensible Markup Language*), integração com tecnologias de componentes distribuídos (ex. CORBA, DCOM) e síntese de hardware e software embarcado.

Ferramentas para co-simulação como Pia [HIN97a] [HIN98] e Polis [BAL97] são implementados sobre domínios (*domains*) do Ptolemy.

3.3.2 IPChinook

O IPChinook [CHO99] é um *framework* voltado para síntese de sistemas embarcados distribuídos e heterogêneos. Permite a modelagem baseada em componentes IP e oferece ferramentas de suporte para projeto, simulação e síntese de sistemas embarcados distribuídos.

No IPChinook, as tarefas de simulação são executadas pela ferramenta Pia [HIN96], a qual dá suporte à utilização de técnicas de simulação que permitem ao projetista

validar seu sistema nos diferentes estágios da síntese, sem a necessidade da construção de um protótipo de hardware. Essa ferramenta permite a especificação de múltiplos modelos de comunicação para cada interface e permite a troca dinâmica entre eles durante a execução da simulação, tornando possível a simulação em diferentes níveis de abstração.

3.3.3 DISCOE

O DISCOE (*DIstributed Simulation COllaborative Environment*) [WIL99] é um projeto iniciado pelo Clifton Labs, Inc. com o objetivo de desenvolver um ambiente distribuído voltado para co-design de hardware e software.

O DISCOE oferece um *backplane* para simulação distribuída e heterogênea que dá suporte a vários modelos de simulação. Ele agrega um conjunto de tecnologias necessárias pra oferecer um ambiente de simulação baseado na *web* com ênfase na preservação da propriedade intelectual (IP).

Na abordagem utilizada pelo DISCOE cada participante na simulação distribuída conhece somente as interfaces externas dos outros participantes. Assim, somente informação de simulação é trocada entre os participantes, de modo que é preservada a propriedade intelectual contida em cada um deles.

3.3.4 Ambiente S^3E^2S

O ambiente S^3E^2S (*Specification, Simulation, and Synthesis of Embedded Electronic Systems*) [WAG99] [OYA99] é construído sobre o simulador SIMOO [COP97], que é um ambiente integrado voltado para modelagem orientada a objetos e simulação de sistemas discretos.

No S^3E^2S, o simulador SIMOO é acoplado ao simulador VSS Synopsys. Esse acoplamento é realizado através da geração automática das interfaces necessárias para as entidades VHDL e as entidades SIMOO. Nessa abordagem, as interfaces geradas são totalmente responsáveis pela troca de dados e sincronização.

A comunicação entre o SIMOO e o VSS é realizada através de *sockets*, o que permite uma solução distribuída. Já a sincronização entre as entidades é obtida através de um algoritmo conservativo.

3.3.6 MCI (Multilanguage Distributed Co-Simulation Tool)

O ambiente MCI [HES99] é voltado para execução e coordenação de simuladores concorrentes para a simulação de diferentes subsistemas. A comunicação entre simuladores é automaticamente gerada a partir de um modelo de configuração. Esse modelo define as interações entre os módulos, que podem ser especificadas em diferentes níveis, desde o nível de implementação até o nível de aplicação, onde a comunicação é executada através de primitivas de alto-nível (ex. *send* e *receive*).

Baseado em informações fornecidas pelo modelo de configuração, o MCI permite a geração de um ambiente de co-simulação multi-linguagem, que é composto de um *backplane* de co-simulação e uma interface de simulação para cada simulador que será

executado. Um aspecto que deve ser destacado na abordagem utilizada pela MCI é a clara separação entre os módulos de comunicação e o restante do sistema.

O ambiente MCI, apesar de oferecer suporte para co-simulação distribuída, não possui um modelo para sincronização dos simuladores (*untimed co-simulation*), assim cada simulador executa sob seu relógio local e a troca de dados é controlada por eventos.

3.3.7 ROSES

ROSES [CES2002] [DZI2003] é uma metodologia de projeto, agregada a um conjunto de ferramentas, voltada para vencer as diferenças entre a especificação de sistemas heterogêneos e a implementação desses sistemas em chips.

O ambiente ROSES integra ferramentas para geração automática de *wrappers* de co-simulação / síntese para componentes de hardware e de software. Os *wrappers* utilizados para co-simulação são responsáveis pela adaptação de diferentes interfaces de simuladores para o barramento de co-simulação e permitem a adaptação de protocolos de comunicação e níveis de abstração. O processo de geração automática dos *wrappers* é realizado a partir de um conjunto de bibliotecas de componentes extensíveis que tornam possível a geração desses *wrappers* tanto para co-simulação quanto para propósitos de síntese, utilizando uma arquitetura comum.

O modelo de arquitetura virtual empregado no ROSES separa claramente o comportamento e a estrutura do componente das primitivas de comunicação, permitindo que o processo de refinamento do componente e da comunicação seja executado em separado e de forma concorrente.

4 ARQUITETURA DCB

A arquitetura DCB (*Distributed Co-Simulation Backbone*) [MEL2001] é resultado de um projeto que está sendo desenvolvido em uma tese de doutorado do PPGC. A tese propõe uma infraestrutura de suporte, inicialmente voltada para sistemas embarcados (*embedded systems*), que forneça serviços de cooperação e comunicação e permita a integração e execução de co-simulações distribuídas e heterogêneas.

Alguns dos conceitos utilizados na definição da arquitetura DCB (tais como federado, federação, gerenciamento de propriedade e outros) são baseados no padrão HLA, abordado no Capítulo 2.3. A incorporação de definições do padrão HLA na arquitetura DCB é um dos principais diferenciais com relação aos mais conhecidos ambientes de co-simulação. Essas características, agregadas ao ambiente de co-simulação, são fundamentais para a diminuição da complexidade originada principalmente pela heterogeneidade e pela distribuição.

O principal desafio na co-simulação é o desenvolvimento de tecnologias e mecanismos suficientemente flexíveis e genéricos que permitam a cooperação entre partes heterogêneas, sem que seja necessária a imposição de padrões proprietários para formatos de dados e comportamento de federados participantes da simulação.

Na co-simulação que envolve federados heterogêneos, uma das principais dificuldades encontradas é lidar de uma forma padronizada com os diferentes padrões de interface e diferentes formas de comunicação que estes federados oferecem. Por isso, a arquitetura DCB foi projetada para oferecer mecanismos genéricos para integração de novos simuladores/modelos de simulação a uma simulação já existente. No entanto, em muitos casos, pode existir a necessidade de que o projetista faça algum tipo de modificação na interface desse novo federado, devido à impossibilidade de prever todos os tipos e formatos de interfaces existentes e que virão a ser desenvolvidos.

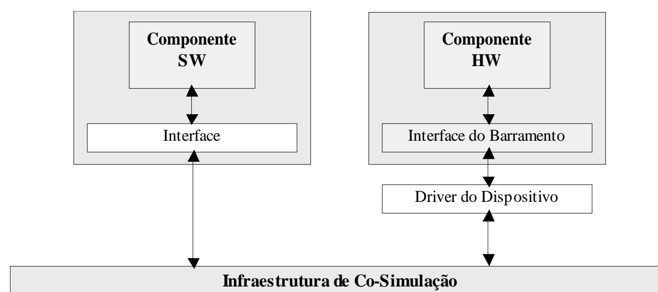


Figura 4.1: Integração de sistemas heterogêneos

Um dos principais objetivos da arquitetura DCB é permitir que federados construídos com diferentes propósitos e com o uso de tecnologias variadas possam cooperar em uma mesma federação com objetivos comuns. A arquitetura DCB oferece suporte à execução de sistemas heterogêneos, neste caso, simulações compostas por partes de hardware (inclusive componentes de hardware reais) ou software (FIGURA 4.1) descritos em diferentes linguagens e/ou em diferentes níveis de abstração.

O desenvolvimento da arquitetura DCB também tem por objetivo oferecer ao projetista a possibilidade da reutilização de federados, de modo que estes, com o mínimo de alterações, possam ser reutilizados em outras simulações ou aplicações. Esse objetivo pode ser alcançado se a infraestrutura de suporte fizer o menor número possível de exigências ao federado que será adicionado à federação.

A abordagem utilizada pela arquitetura DCB permite que a modelo da co-simulação (federação) seja construído através da integração de componentes modulares, que assim como em HLA, são chamados de federados. Essa abordagem, que é semelhante à utilizada na área de componentes de software, exige que o federado disponibilize somente a sua interface de comunicação, para tornar possível a comunicação com a arquitetura DCB. Assim, aspectos da implementação interna do federado são deixados de lado e a maior preocupação do projetista torna-se a correta cooperação entre os federados. Essas características fazem com que os federados que estão participando da simulação sejam comparados a componentes *black-box*.

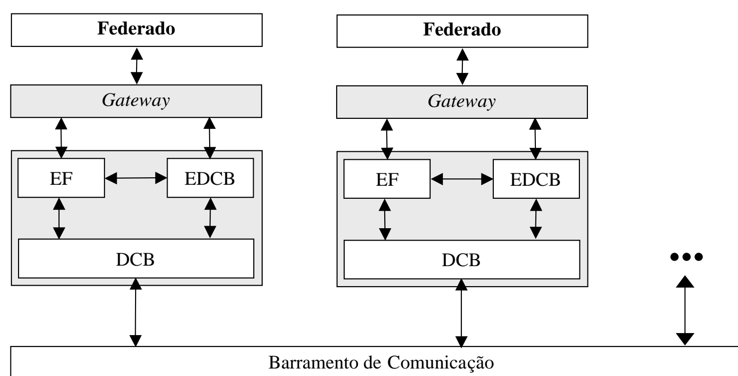


Figura 4.2: Arquitetura DCB (*Distributed Co-Simulation Backbone*)

A arquitetura DCB também herda do padrão HLA o conceito de embaixadores, que são entidades que oferecem serviços de comunicação entre o DCB e o federado. A FIGURA 4.2 apresenta a arquitetura DCB, que está logicamente dividida em três camadas: (1) a camada DCB, (2) os embaixadores (Embaixador Federado - EF e Embaixador DCB - EDCB) e (3) o *gateway*.

Apesar de ser baseada em HLA, a arquitetura DCB não oferece suporte automático à integração de federados desenvolvidos/modificados especificamente para serem compatíveis com HLA. Para tornar uma simulação compatível com HLA é necessário, entre outras tarefas, incluir no comportamento do modelo chamadas diretas aos diversos serviços disponibilizados pelo RTI HLA. Segundo Mello [MEL2002] essa é uma das principais desvantagens do padrão HLA, pois existe a necessidade de modificar o código fonte do modelo, limitando a utilização de vários tipos de componentes que não oferecem código fonte por questões de propriedade intelectual.

No contexto de co-simulação de sistemas embarcados, a arquitetura DCB pode ser considerada uma alternativa mais flexível que a HLA, pois não exige que os federados façam chamadas diretas aos serviços do DCB. Na arquitetura DCB o federado precisa somente enviar, quando conveniente, as atualizações de valores de atributos utilizando um dos meios disponibilizados pelo gateway. Assim, é possível que simuladores (ou modelos) heterogêneos sejam adicionados a uma simulação já existente sem impor restrições drásticas ou alterações internas nesses componentes. As principais exigências para que um federado possa ser integrado à arquitetura DCB são:

1. Federado deve possuir uma interface definida e publicamente acessível;
2. Federado deve enviar ao *gateway* atualizações dos valores dos seus atributos (quando for conveniente), através de umas das diversas formas oferecidas (ex. chamada de função, *sockets*, arquivos);
3. Federado deve disponibilizar o seu tempo local (LVT) como um atributo de interface;
4. Federados que executam de forma assíncrona devem permitir que o estado dos objetos do federado seja restaurado através da interface, de modo que a infraestrutura de co-simulação possa gerenciar a execução de rotinas de *rollback*, baseadas no histórico armazenado (*checkpoints*).

Conforme mostra a FIGURA 4.3, a arquitetura DCB foi projetada de forma modular, oferecendo uma clara separação entre as funcionalidades de cada camada. Essa característica facilita possíveis substituições de camadas, permitindo ao desenvolvedor criar diferentes composições para o ambiente de simulação. Nessa arquitetura, os embaixadores e a camada DCB são responsáveis pelas tarefas que envolvem a distribuição e a sincronização entre os componentes participantes da co-simulação. Já o *gateway*, em conjunto com os procedimentos da interface do federado, é responsável pelas rotinas de integração de linguagens e adaptação de interface. A arquitetura em camadas também possibilita, quando necessário, a agregação de *wrappers* que tornam possível a adaptação de protocolos ou encapsulamento da interface real do federado.

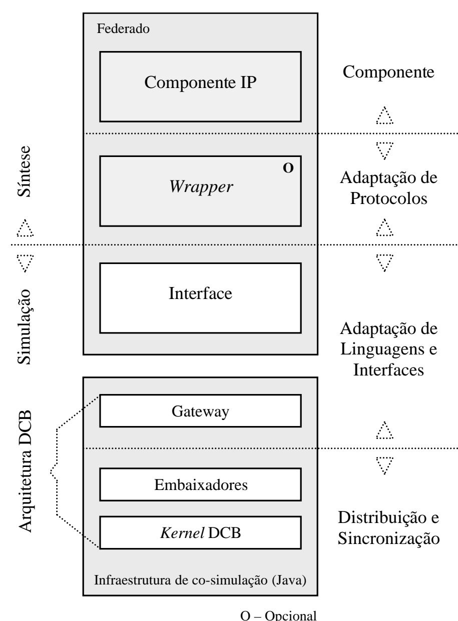


Figura 4.3: Visão funcional da arquitetura DCB

Levando em conta a heterogeneidade dos federados que podem cooperar em um ambiente de co-simulação, a implementação de uma interface de comunicação torna-se uma tarefa complexa [MEL2002]. Com o objetivo de tratar esta complexidade mantendo bons níveis de flexibilidade na agregação de novos federados, a arquitetura DCB utiliza o conceito de *gateway* [STR98]. Apesar da inexistência de uma metodologia para modelagem de federados voltados para a arquitetura DCB, a observância de um determinado aspecto no momento da construção dos federados pode facilitar a tarefa de integração dos mesmos ao *gateway*. Esse aspecto está relacionado à construção do federado de modo que seja mantida uma clara separação entre a sua interface e o seu comportamento. Dessa forma, é possível minimizar o esforço a ser despendido na integração da interface ao *gateway*, já que essa abordagem não exige qualquer modificação no comportamento do federado.

A separação entre interface e comportamento também facilita a substituição do protocolo da interface do federado, tornando mais rápido o processo de refinamento da comunicação, que normalmente é necessário quando um federado é utilizado em diferentes níveis de abstração de um mesmo modelo.

Para diminuir a complexidade das tarefas relacionadas à distribuição e sincronização, a arquitetura DCB utiliza dois embaixadores distintos, fornecendo um embaixador para tratar da interface do federado (Embaixador Federado – EF) e outro para tratar da interface com a camada DCB (Embaixador DCB – EDCB). Deve-se ressaltar que cada federado envolvido na simulação possui, de forma dedicada, o seu DCB e seus embaixadores, sendo que o código destes é exatamente igual em todos os nodos da simulação.

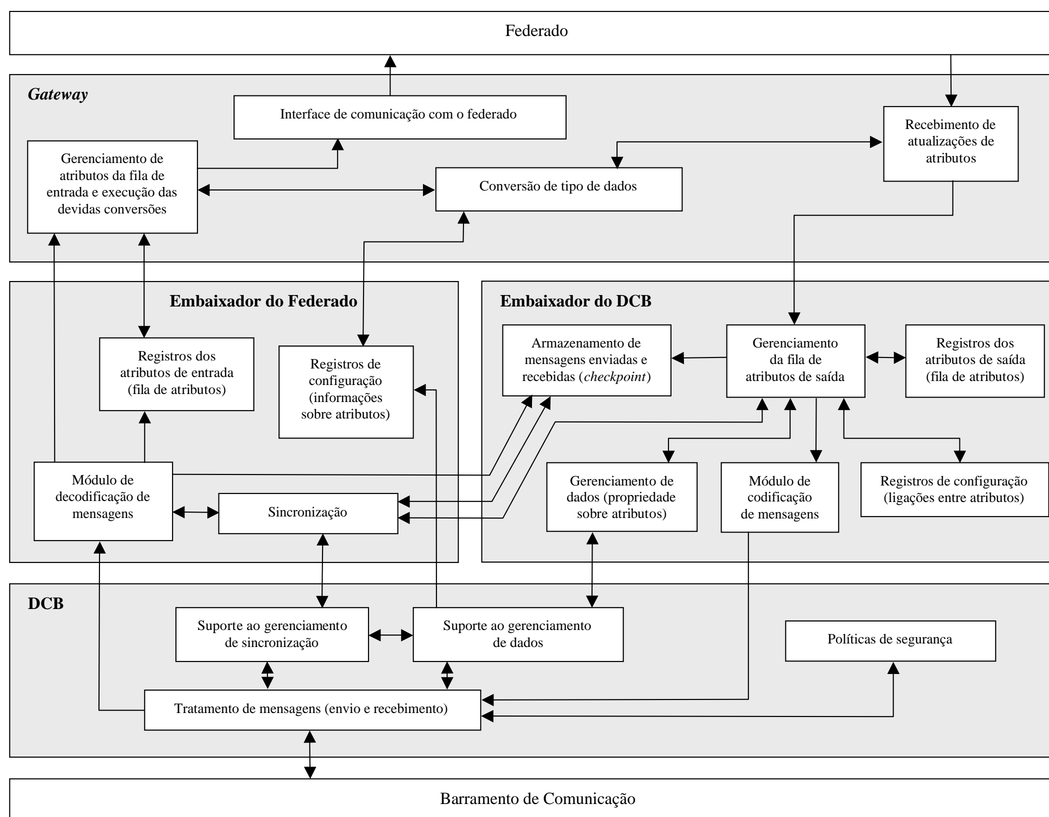


Figura 4.4: Detalhes internos da arquitetura DCB

A FIGURA 4.4 mostra em detalhes os módulos internos de cada uma das camadas que compõe a arquitetura DCB. As próximas seções fazem uma especificação detalhada de cada uma delas.

4.1 Distributed Co-Simulation Backbone (DCB)

O DCB pode ser visto como uma camada específica de simulação para suporte à co-simulação distribuída. Funcionalmente, é bastante similar ao RTI da arquitetura HLA [MEL2001]. Como pode ser visto na FIGURA 4.5, seu principal objetivo é oferecer mecanismos genéricos de suporte para prover serviços de cooperação, comunicação e sincronização entre federados heterogêneos.

Como foi citado na seção anterior, cada federado possui uma camada DCB dedicada, assim cada nodo participante da simulação atua tanto como cliente quanto como servidor, dependendo da direção dos fluxos de dados e do recebimento de informações de outros federados. A inexistência de uma entidade que centraliza o controle das trocas de dados caracteriza a arquitetura DCB como *peer-to-peer* [ROD2003], trazendo vantagens em relação a escalabilidade e tolerância a falhas. Desse modo, situações como a queda de um nodo ou a inclusão de um novo componente não implicam em grandes prejuízos ou modificações à federação.

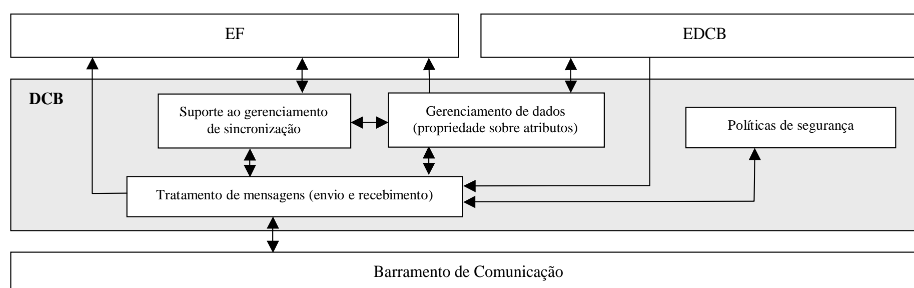


Figura 4.5: Arquitetura da camada DCB

A camada DCB é dividida em quatro módulos:

1. Tratamento de mensagens – esse módulo é responsável pela efetiva comunicação entre os federados participantes da simulação. Ele cuida do recebimento e tratamento de novas conexões solicitadas por outros federados, assim como do envio de mensagens de acordo com as solicitações dos módulos de gerenciamento de dados, sincronização e do módulo de codificação de mensagens do EDCB. A troca de mensagens entre federados é feita de forma padronizada, como pode ser visto no Capítulo 5.1.

2. Suporte ao gerenciamento de sincronização – a principal tarefa desse módulo é o gerenciamento do GVT. Esse módulo também é responsável pelo tratamento de anti-mensagens (envio e recebimento). Quando uma

anti-mensagem é recebida, o que indica que o federado deve voltar atrás no seu tempo (*rollback*), o módulo de sincronização do EF é imediatamente notificado, podendo este inicializar os mecanismos de *rollback* e possivelmente solicitar o envio de anti-mensagens para outros federados.

3. Suporte ao gerenciamento de dados – Esse módulo tem a tarefa de coordenar o gerenciamento de propriedade, mantendo uma política de exclusão mútua sobre o uso de cada um dos atributos dos federados (válido somente para atributos de entrada). Assim, pode-se garantir que somente um federado possui a propriedade de um atributo em um determinado instante de tempo. No entanto, essa propriedade é dinâmica e pode ser modificada no decorrer da simulação. A informação sobre a propriedade dos atributos de um federado está armazenada no módulo de registro de configuração do EF. Está previsto em [MEL2001] a inclusão de um conjunto de regras (fornecidas pelo projetista), chamadas de regras de cooperação, para facilitar a coordenação da troca de propriedade sobre atributos no decorrer da execução. Também está prevista a inclusão de políticas de replicação de dados, compartilhando informações sobre propriedade em todos os nodos DCB, com o objetivo de agilizar o processo de gerenciamento de propriedade sobre atributos.

4. Políticas de segurança – esse módulo foi previsto na camada DCB para cuidar do gerenciamento de erros gerados por situações adversas como, por exemplo, a queda de um dos nodos participantes da simulação.

O principal foco dessa dissertação está relacionado com os embaixadores e o *gateway*. Entretanto, para tornar possível a validação dos mesmos foi necessária a especificação do módulo de tratamento de mensagens na camada DCB. Já a implementação dos demais módulos dessa camada é o principal objetivo da tese de doutorado [MEL2001] que está em andamento.

4.2 Embaixadores

Os embaixadores, em conjunto com o *gateway*, são as entidades responsáveis pelas interfaces de comunicação entre o federado e a federação. A definição e a construção dos embaixadores é essencial para que o DCB possa suportar adequadamente a execução de um modelo de co-simulação [MEL2001].

A arquitetura DCB foi projetada para dar suporte à utilização de dois embaixadores: o embaixador do DCB (EDCB) e o embaixador do federado (EF). Em uma visão de alto nível, pode-se dizer que o EDCB oferece serviços padronizados para comunicação com o DCB e o EF oferece serviços para comunicação com o *gateway* / federado.

Mello [MEL2001] afirma que a geração automática dos embaixadores dos federados é bastante desejável devido à potencial redução do tempo de implementação de um modelo de co-simulação. No entanto, com o desenvolvimento de protótipos, notou-se a viabilidade de criar embaixadores genéricos e configuráveis.

O processo de configuração dos embaixadores é realizado de forma automática e sem a necessidade de recompilar código. Para auxiliar na tarefa de configuração é descrita no Capítulo 5.6 uma ferramenta desenvolvida para esse propósito.

A seguir são descritos em detalhes os dois embaixadores e suas principais funções dentro da arquitetura DCB.

4.2.1 Embaixador do DCB (EDCB)

O embaixador do DCB (EDCB), detalhado na FIGURA 4.6, é responsável por tarefas que envolvem gerenciamento de propriedade sobre atributos, armazenamento do histórico de valores enviados e recebidos (*checkpoint*) no decorrer da execução, gerenciamento dos atributos de saída recebidos do federado, empacotamento de mensagens e gerenciamento da estrutura de dados que armazena as ligações entre os atributos de saída do federado que representa e os atributos de entrada de outros federados (inclusive de outras federações).

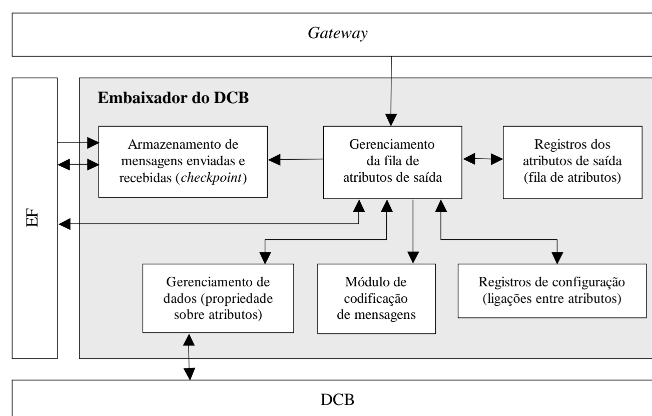


Figura 4.6: Arquitetura do Embaixador do DCB (EDCB)

O EDCB é composto por seis principais módulos:

1. Gerenciamento da fila de atributos de saída – Esse módulo é responsável por todas as operações envolvidas no gerenciamento da fila de atributos recebidos do federado. Isso inclui tarefas como:

    (a) Inclusão de atributos – quando o *gateway* solicita uma atualização de atributo, esse módulo é responsável pela inclusão, na fila de atributos de saída, de uma entrada para cada um dos atributos destino, de acordo com os registros de configuração do embaixador;

    (b) Exclusão de atributos na fila – um atributo é excluído da fila quando é recebida uma notificação indicando que a propriedade sobre o determinado atributo destino foi conseguida. Assim, a mensagem de atualização é codificada, enviada para o barramento de comunicação e o atributo pode ser excluído da fila;

(c) Solicitação de propriedade – é responsável indireto pela solicitação de propriedade sobre determinado atributo destino. A cada atributo incluído na fila é feita uma notificação para o módulo de gerenciamento de propriedade, de forma que este é o responsável efetivo pela solicitação da propriedade;

2. Registros dos atributos de saída – é responsável pelo armazenamento dos atributos recebidos do federado em uma fila. Esses atributos são originários de operações de atualização de valores e ficam armazenados na fila enquanto não for possível obter a propriedade sobre o atributo destino;
3. Gerenciamento de dados – trata do gerenciamento de propriedade para cada ligação (*link*) entre um atributo origem e um atributo destino. Quando é recebida uma mensagem indicando a propriedade sobre determinado atributo destino, o módulo de gerenciamento da fila é notificado, indicando que o atributo pode ser enviado para o seu destino. Como citado anteriormente, o gerenciamento será realizado através de regras definidas pelo projetista e fornecidas com o modelo da federação;
4. Codificação de mensagens – é responsável pelo empacotamento das mensagens que serão repassadas à camada DCB. As mensagens são codificadas pelo EDCB em um formato padronizado e agregam informações como: federação fonte, federado fonte, federação destino, federado destino, LVT da fonte, id do atributo, valor do atributo e outras. Essas informações são necessárias para que a comunicação entre federado origem e federado destino se realize corretamente. Por isso, os registros de configuração e outras informações mantidas pelo EDCB são essenciais para o envio e recebimento de mensagens;
5. Armazenamento de mensagens enviadas e recebidas – módulo responsável pela estrutura de dados para armazenar os estados do federado (*checkpoint*) no decorrer de sua participação na co-simulação. Esses *checkpoints* serão futuramente utilizados para permitir operações de *rollback* (restauração de estados seguros) em federados que executam de forma assíncrona. Entretanto, a implementação de mecanismos de *rollback* e controle de *checkpoints* implica no estudo detalhado de algoritmos específicos para estes fins [MEL2001];
6. Registros de configuração – módulo responsável pela estrutura de dados que armazena as ligações entre os atributos de saída do federado que representa e os atributos de entrada de outros federados (inclusive de outras federações). As ligações entre todos os federados definem o comportamento geral do modelo da federação. Devido ao modo como a arquitetura foi projetada (*peer-to-peer*), não existe a necessidade de todos os federados armazenarem no seu registro de configuração as ligações de todos os federados. Somente é necessário que o EDCB conheça as informações sobre as ligações com os federados com os quais ele possui relação. Conforme mostra a FIGURA 4.7, um federado A não precisa conhecer que tipo de cooperação é realizada entre dois outros federados B e C [MEL2001].

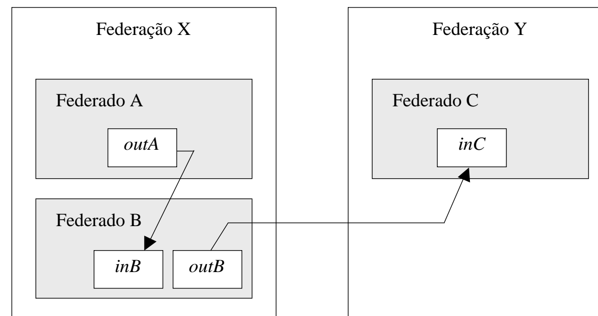


Figura 4.7: Ligação entre atributos de federados (configuração da federação) (Fonte: [MEL2001])

Um ponto que deve ser levado em consideração no módulo de armazenamento de mensagens enviadas e recebidas é o consumo de memória excessivo ocasionado pelo armazenamento desses registros. Para amenizar esse problema podem ser utilizados métodos como o *fossil collection* [FER95], que está relacionado com a eliminação definitiva de eventos que possuem um *timestamp* menor que o registrado pelo GVT.

4.2.2 Embaixador do Federado (EF)

O EF é o módulo responsável pelo registro dos atributos de entrada e saída (nome, tipo e propriedade) que compõem a interface do federado a que pertence. Além disso, o EF também trata de aspectos de sincronização e manutenção de valores de atributos, que são recebidos do DCB e armazenados em uma fila de atributos.

Os atributos da interface do federado podem ser classificados em três tipos: *read* (somente leitura), *write* (somente escrita) e *read*/*write* (leitura e escrita). Além disso, um valor de um atributo na interface do federado pode ser constante ou variável. Por exemplo, o atributo definido pelo projetista que especifica o modo de execução do federado (síncrono ou assíncrono) pode ser considerado um valor constante, pois não será alterado no decorrer da simulação.

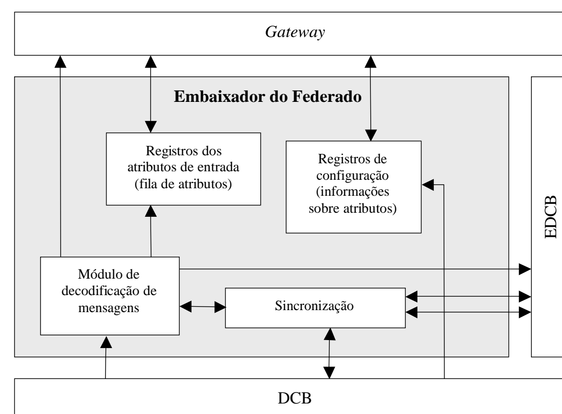


Figura 4.8: Arquitetura do Embaixador do Federado (EF)

O embaixador do federado, como pode ser visualizado na FIGURA 4.8, é composto por quatro principais módulos:

1. Módulo de decodificação de mensagens – é responsável pela decodificação de mensagens recebidas do DCB. O valor do atributo recebido na mensagem é armazenado na fila de atributos do EF. Esse módulo possui mais duas tarefas: solicitar ao EDCB o armazenamento das atualizações junto ao histórico (*checkpoint*) e notificar o *gateway* quanto ao recebimento de uma atualização de atributo;

2. Registro dos atributos recebidos – esse módulo é composto pela fila de atributos e por mecanismos que permitem o seu gerenciamento. Essa fila armazena id do atributo, valor do atributo, LVT do emissor, identificação do emissor;

3. Registros de configuração – são inicializados com as informações referentes a cada um dos atributos da interface do federado (fornecidos pelo projetista). Armazena dados como nome, tipo e proprietário do atributo. Esse módulo oferece às outras camadas um serviço de informação sobre tipos de atributos, para fins de conversão;

4. Sincronização – esse módulo disponibiliza mecanismos para sincronização tanto para federados que executam de forma síncrona quanto assíncrona. Ele é responsável pela detecção de situações de inconsistência na sincronização, que podem ser causadas por vários motivos, como violação da LCC, erros na construção dos modelos, situações não previstas, falhas de comunicação. Em alguns desses casos é necessária a execução de mecanismos de *rollback*, de modo que o federado volte a um estado consistente. O *rollback* é necessário em dois casos:

    (a) Quando ocorre uma violação à LCC, ou seja, quando é armazenado na fila de entrada um atributo com *timestamp* menor do que o LVT (*Local Virtual Time*) corrente;

    (b) Quando o módulo de suporte à sincronização (DCB) recebe anti-mensagens de outros federados, indicando que o federado deverá voltar no tempo (*rollback*) para manter a simulação consistente.

Uma das exigências para integração de federados que executam de forma assíncrona (citada no Capítulo 4) é que o federado permita que o estado dos seus objetos seja restaurado através da sua interface. Assim, a operação de *rollback* é realizada em três passos:

1. Busca dos valores dos atributos (no determinado instante de tempo) armazenados no histórico do EDCB (*checkpoint*);
2. Atribuição desses valores aos atributos de entrada do federado;
3. Verificação de registros de valores de atributos enviados a outros federados no intervalo de tempo que o federado estava em estado inconsistente e

imediata solicitação de envio das respectivas anti-mensagens, indicando que esse(s) federado(s) também deverá(ão) iniciar um procedimento de *rollback*.

O envio de anti-mensagens, em muitos casos, resultará em sucessivas operações de *rollback* nos outros participantes da federação, o que implica em mais anti-mensagens circulando pelo barramento de comunicação, podendo comprometer o desempenho do sistema.

No EF, além das informações sobre os atributos de entrada e saída disponibilizados pela interface do federado, são armazenadas informações como identificação do federado, tempo local (LVT), modo de execução (síncrono/assíncrono) e outros.

A identificação do federado (*federateid*) é definida no ambiente de modelagem, no entanto, o único cuidado que deve ser tomado é que esta identificação deve ser única dentro de uma federação. Assim, mesmo que existam dois federados replicados em uma mesma federação, eles serão tratados de forma independente pelo DCB [MEL2001].

A origem das mensagens trocadas entre os federados que estão cooperando através do DCB é definida pela identificação do federado (*federateid*) em conjunto com a identificação da federação (*federationid*). O mesmo vale para a definição do destino das mensagens. As identificações únicas para federado e federação são necessárias para permitir a cooperação entre federados pertencentes a diferentes federações, de modo que estes sempre mantenham uma identificação única.

4.3 Gateway

A camada *gateway*, ilustrada na FIGURA 4.9, pode ser considerada a interface da arquitetura DCB. Para que um federado possa participar de uma simulação utilizando os serviços da arquitetura DCB é necessário que ele conheça a interface disponibilizada pelo *gateway*. Da mesma forma, é necessário que o *gateway* conheça a interface disponibilizada pelo federado.

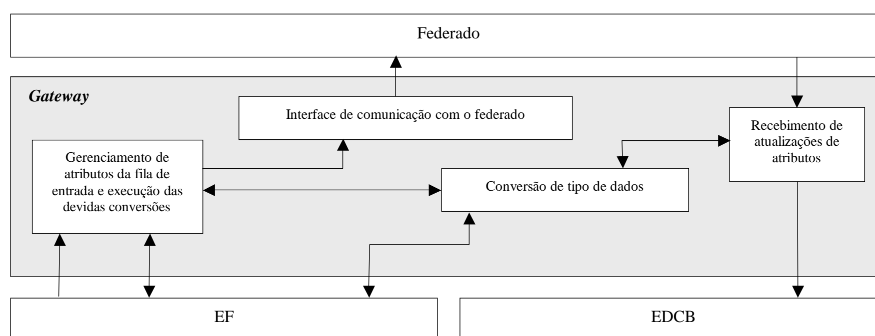


Figura 4.9: Arquitetura do *Gateway*

A construção da interface de comunicação é um dos pontos mais críticos na implementação de uma co-simulação. Segundo Mello [MEL2001], quando um novo federado for adicionado a uma federação, a construção de um *gateway* evita possíveis

modificações na infraestrutura de co-simulação, no entanto não diminui o esforço de implementação para integrar esse federado ao *gateway*.

O *gateway* é responsável pela execução de quatro principais tarefas, conforme detalhado abaixo:

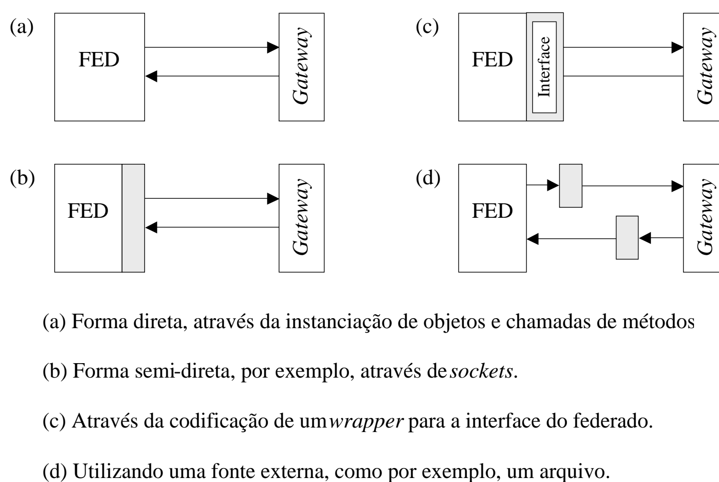
1. Receber do federado atualizações de valores de atributos. As formas para o recebimento das atualizações podem variar de acordo com o *template* utilizado para gerar o *gateway*. Por exemplo, um federado escrito em Java simplesmente fará uma chamada de função, já um federado remoto deverá enviar uma string via *sockets* TCP;
2. Executar a conversão de tipo dos dados recebidos. Para executar essa tarefa são buscadas informações sobre o tipo de dado destino junto aos registros de configuração do embaixador do federado, que armazenam as informações sobre os atributos da interface;
3. Executar a conversão semântica dos dados recebidos, quando necessário. Esse tipo de necessidade pode surgir quando é preciso agregar à co-simulação federados modelados em alto nível de abstração. Essa conversão, que pode ser chamada de mapeamento atributo-valor, consiste na conversão de um ou mais atributos recebidos em uma chamada de função do federado que possui um ou mais parâmetros associados;
4. Receber da infraestrutura de co-simulação notificações referentes à chegada de atualizações de valores de atributos do federado, buscar esses valores na fila de entrada do embaixador do federado e entregá-los à interface de comunicação do federado. Ainda é necessário eliminar os valores de atributos já entregues ao federado, da fila de entrada do EF.

Foi convencionado em [MEL2001] que a tradução dos dados recebidos do federado deverá ser sempre para o tipo *String*. A especificação da arquitetura DCB define o tipo *String* como padrão para troca de dados entre as camadas da arquitetura DCB. A escolha de um tipo único para trocas de dados internas facilita a implementação dos embaixadores e evita subseqüentes conversões de dados entre as camadas da arquitetura.

A arquitetura do *gateway* foi projetada com o objetivo de evitar possíveis modificações na infraestrutura de co-simulação quando um novo federado for adicionado à federação. No entanto, isso não diminui o esforço de implementação para integrar esse federado ao *gateway*. Para diminuir a complexidade dessa tarefa, é possível gerar o *gateway* de forma automática, com o auxílio de *templates* pré-definidos. Isso pode ser feito com federados que possuem um conjunto previsto de características. Contudo, é muito difícil manter a generalidade quando não existe previsão (ou definição) das características de um novo federado [MEL2001].

O *template* do *gateway* sofre alterações em função do tipo de interface oferecida pelo federado que está sendo adicionado à federação. Por isso, em muitos casos, o *gateway* pode possuir funcionalidades específicas, desenvolvidas para atender aos mais diversos tipos de interfaces existentes.

Para definir diferentes *templates* de *gateway*, foi necessária a identificação das diferentes alternativas para a implementação da comunicação entre o *gateway* e o federado. A FIGURA 4.10 ilustra as principais formas para implementação da comunicação.



(a) Forma direta, através da instanciação de objetos e chamadas de métodos

(b) Forma semi-direta, por exemplo, através de *sockets*.

(c) Através da codificação de um *wrapper* para a interface do federado.

(d) Utilizando uma fonte externa, como por exemplo, um arquivo.

Figura 4.10: Formas para implementar a comunicação entre o *gateway* e o federado

Para este trabalho, foi prevista a especificação de três *templates* diferentes, para atender as alternativas levantadas na FIGURA 4.10. Esses *templates* estarão disponíveis junto à ferramenta de configuração (descrita na Seção 5.6) e serão configurados de forma automática, de acordo com a especificação fornecida pelo projetista. A especificação detalhada de cada um deles pode ser encontrada na Seção 5.5.

4.4 Inicialização, controle e acompanhamento de experimentos de co-simulação utilizando a arquitetura DCB

O principal objetivo da co-simulação é permitir a investigação do comportamento do sistema. Para isso, muitas vezes a co-simulação exige do sistema que está sendo construído funcionalidades além das necessárias na implementação final. Essas funcionalidades não são sintetizáveis e são agregadas ao sistema para, por exemplo, permitir a execução de testes ou visualização dos estados internos das entidades.

Um dos objetivos do desenvolvimento da arquitetura DCB é prover mecanismos para evitar modificações desnecessárias no código dos federados. Assim, funções como visualização, execução de testes e gerenciamento da execução da simulação devem ser separadas do modelo em si e implementadas em federados específicos. Esses federados, dependendo do domínio da aplicação e da funcionalidade requerida pelo projetista, podem ser passivos ou ativos, ou seja, interferem ou não na co-simulação.

A implementação de federados genéricos e configuráveis que executem funções específicas (coleta de dados, visualização de dados e outros) facilita o reuso em diferentes situações. Esses federados podem ser inseridos na federação da mesma forma que ocorre com federados tradicionais. Assim, o projetista poderia buscar federados já existentes na sua biblioteca, agregá-los à federação utilizando recursos da ferramenta de

modelagem, configurá-los de acordo com as suas necessidades e, finalmente, fazer as ligações com os atributos de entrada e/ou de saída de outros federados.

Outro aspecto que precisa ser levado em consideração é a inexistência no projeto da arquitetura DCB de uma entidade central para coordenação, inicialização e gerenciamento dos federados. Por isso, é fundamental que uma ou mais entidades participantes da federação sejam responsáveis pelas tarefas de inicialização (disparo), controle, monitoramento, ou interrupção abrupta da execução da federação. Para isso existem duas principais alternativas: (a) a criação de federados genéricos e configuráveis voltados para a execução dessas tarefas ou (b) estender a arquitetura DCB, de forma que esta ofereça uma entidade central para coordenação da federação.

Na especificação atual da arquitetura DCB não estão previstas essas características, desse modo, a inicialização do federado fica a cargo do *gateway*. Para isso, a interface do federado deverá ser construída de forma a oferecer um método para a inicialização do mesmo. A disponibilização dessa característica na interface do federado permite que a execução seja iniciada remotamente através do recebimento de um valor de atributo de outro federado ou futuramente até do próprio DCB, se for necessário.

5 IMPLEMENTAÇÃO

Os protótipos foram implementados em Java, utilizando técnicas de orientação a objetos. A escolha da linguagem Java se deve principalmente ao fato de que Java é independente de plataforma, o que é um aspecto importante, pois o ambiente deverá atender a federados desenvolvidos nas mais diversas plataformas. Além disso, as rotinas nativas da linguagem para comunicação e tratamento de XML também foram decisivas na escolha.

Os protótipos desenvolvidos, bem como o modelo utilizado para o estudo de caso, são detalhadamente descritos nas seções seguintes.

5.1 Arquitetura DCB

A implementação do protótipo da arquitetura DCB (FIGURA 5.1) segue, em linhas gerais, as descrições do Capítulo 4 e está mais bem detalhada nas próximas seções. Todas as camadas foram construídas de forma modular, de modo que sejam facilitadas possíveis extensões. Essas extensões incluem os módulos necessários para implementar a sincronização e o gerenciamento de dados (gerenciamento de propriedade) que são o principal foco da tese de doutorado em andamento (citada anteriormente).

O barramento de comunicação da arquitetura DCB foi implementado de modo que fosse possível a cooperação transparente, em uma mesma federação, entre federados distribuídos (executando em máquinas distintas) e federados locais (executando em diferentes *threads* de uma mesma JVM). O mecanismo para comunicação entre federados locais está implementado nas classes DCB e *DCBMThread*, já o mecanismo para comunicação entre federados remotos é de inteira responsabilidade da classe DCB. A decisão sobre o método de envio da mensagem é tomada no módulo *DCBSend* da camada DCB, que automaticamente identifica se o endereço do destinatário é remoto ou local.

As tarefas de inicialização e configuração da infraestrutura DCB (*gateway*, embaixadores e camada DCB) são de responsabilidade das classes *ApplicationDCB* e *DCBMThread.* A classe *ApplicationDCB*, é responsável pela instanciação dos objetos da arquitetura DCB e pela parametrização dos embaixadores e da camada DCB. O construtor da *thread ApplicationDCB* recebe como parâmetro o nome do arquivo de configuração (XML) que contém as informações sobre o federado, faz a leitura do arquivo e instancia os objetos da arquitetura DCB com os seus devidos parâmetros de configuração.

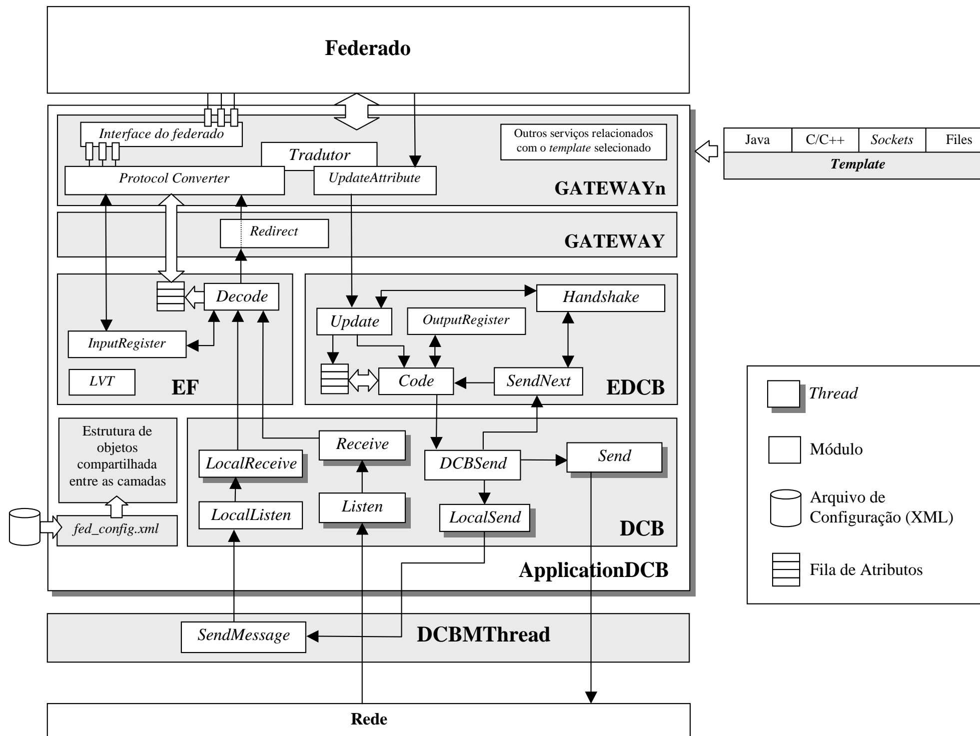


Figura 5.1: Protótipo da arquitetura DCB

Já a classe *DCBMThread* é responsável pela instanciação de uma ou mais *threads* da classe *ApplicationDCB*, que engloba toda a infraestrutura de co-simulação. A *DCBMThread* armazena um índice para todas as *threads* instanciadas (uma *thread* para cada federado sob a mesma JVM) e disponibiliza o método *SendMessage*, que é responsável pela identificação da *thread* destinatária e pelo envio da mensagem.

O mecanismo para comunicação local foi idealizado para evitar a sobrecarga gerada pela comunicação via rede, para casos onde existe a possibilidade e/ou necessidade de executar dois ou mais federados em uma mesma máquina. Esse mecanismo é baseado na utilização de múltiplas *threads* (uma para cada federado), o que exigiu a definição de métodos na classe *ApplicationDCB*, para parametrização das camadas que compõem a arquitetura DCB, pois cada uma das *threads* precisa manipular um conjunto de informações (configurações) de um federado distinto.

A execução de federados em *threads* concorrentes exige a construção de uma classe *gatewayn* (n identifica o federado que o *gateway* representa) diferente para cada federado instanciado sob a mesma JVM. Além disso, foi necessária a construção de uma classe *gateway* para fazer o redirecionamento das chamadas (recebidas do EF) para o *gatewayn* do respectivo federado. A utilização de uma classe *gatewayn* para cada federado é necessária, pois o *gatewayn* deve possuir em seu código a definição dos métodos da interface do federado a quem representa, funcionalidade bastante complexa para ser obtida através de parametrização.

5.2 DCB (*Distributed Co-Simulation Backbone*)

O DCB tem como principal objetivo oferecer serviços de cooperação, comunicação e sincronização entre os federados. A implementação dessa camada não é um dos objetivos desse trabalho. No entanto, para que o estudo de caso fosse possível, foi necessária a implementação de uma versão inicial do DCB (veja FIGURA 5.2), responsável por oferecer apenas primitivas básicas para comunicação local e serviços para comunicação via rede.

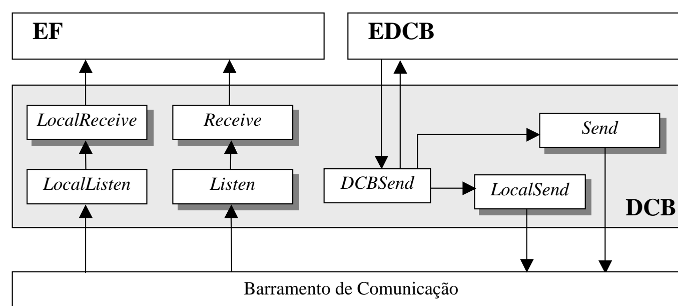


Figura 5.2: Protótipo da camada DCB

O módulo *DCBSend* é responsável por receber atualizações de valores de atributos do EDCB e decidir qual mecanismo de comunicação será utilizado para enviar essas atualizações ao federado destino. Essa decisão é tomada com base no atributo "ip destino" que é armazenado nas configurações do EDCB. Assim, se o valor do atributo "ip" for igual a "local", o *DCBSend* cria uma *thread LocalSend*, responsável por repassar a mensagem ao método *SendMessage* do *DCBMThread.* Esse método identifica a *thread* destinatária da mensagem e faz a entrega da mesma através da chamada ao método *LocalListen.* O *LocalListen* cria uma *thread LocalReceive* que entrega a mensagem diretamente ao módulo *Decode* do EF.

No entanto, se o atributo "ip" do federado destino for um endereço IP válido, o *DCBSend* verifica, junto ao EDCB, se já existe uma referência à *thread Send* associada ao federado destino. Se a referência já existir, a *thread* previamente criada é utilizada para enviar a mensagem ao federado (executando somente as tarefas relacionadas nos itens 2 e 3 descritos abaixo), caso contrário, o *DCBSend* executa as seguintes tarefas:

1. Cria uma *thread Send*, responsável pelo estabelecimento da conexão com o federado destino. No destino, a conexão é recebida pela *thread Listen*, que automaticamente cria uma *thread Receive* para manipular a conexão;
2. Envia a mensagem através da *thread Send*. Essa mensagem é recebida e tratada no destino pela *thread Receive*, previamente criada pela *thread Listen* no momento da conexão;
3. As *threads Send* (do federado emissor) e *Receive* (do federado receptor) são colocadas em modo de espera, mantendo o canal de comunicação aberto;
4. É armazenada uma referência à *thread Send* nos registros do embaixador do DCB do emissor, de modo que seja possível utilizar a mesma *thread* para o envio das mensagens seguintes.

O mecanismo de comunicação desenvolvido para troca de mensagens entre federados remotos é baseado na técnica de serialização de objetos sobre *sockets* Java, que permite o envio da instância de um determinado objeto via *sockets*, através de chamadas de alto nível. A abordagem utilizada na implementação é baseada na criação dinâmica de um conjunto de *threads* (*send* e *receive*) para cada sentido da comunicação com cada um dos federados remotos, permitindo que dois federados comuniquem-se de forma bidirecional ou unidirecional. Essa característica pode definir federados como ativos ou passivos, conforme pode ser visto nos federados B e C da FIGURA 5.3. Além disso, o processo de criação dessas *threads*, descrito anteriormente em quatro passos, somente é necessário na primeira tentativa de envio de mensagem para cada federado, ou seja, após o estabelecimento da primeira conexão é eliminada a necessidade de criação de uma nova conexão para cada tentativa de envio de uma mensagem.

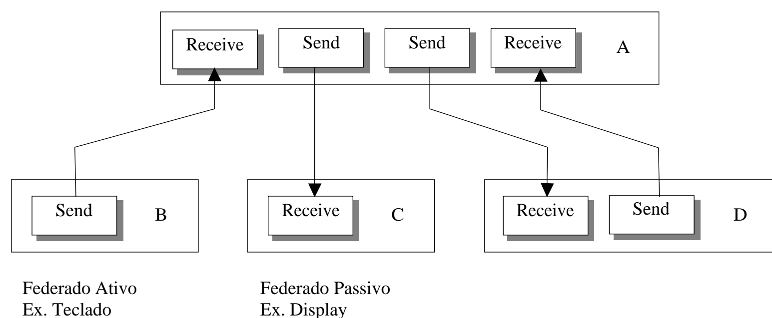


Figura 5.3: Modelo para comunicação remota implementado no DCB

5.3 Embaixador do DCB (EDCB)

O EDCB tem a função de receber dados do federado e encaminhá-los para os respectivos destinos, de acordo com a configuração armazenada. Entretanto, esses dados não podem ser enviados em qualquer instante de tempo, pois deve ser levado em conta que a infraestrutura DCB é distribuída, e por isso podem ocorrer atrasos na entrega de mensagens, troca de ordem no recebimento e, até mesmo, perda destas mensagens.

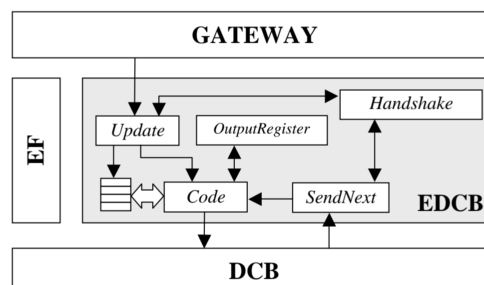


Figura 5.4: Protótipo do embaixador do DCB (EDCB)

Para tratar esses problemas, existe a necessidade da utilização de mecanismos de sincronização que garantam que as mensagens sejam entregues na ordem e no momento

correto, de forma que seja mantida a consistência na execução dos federados participantes da simulação.

Nesse protótipo, foram implementados três principais módulos:

- *Update* – responsável por receber as atualizações de atributos enviadas pelo *gateway*. Esse módulo busca informações sobre nome, tipo de dado e lista de destinos registrados nos objetos *OutputRegister*, e cria para cada atributo destino uma nova entrada na fila de saída. O módulo também interage com o mecanismo de *handshake* (descrito a seguir) para verificar se é possível enviar uma nova mensagem. Caso seja possível, a mensagem é encaminhada para o módulo *Code*.
- *Code* – responsável pela montagem das mensagens que são enviadas para o DCB e a exclusão do registro na fila de saída.
- *SendNext* – esse módulo é acionado quando é confirmado o recebimento de uma mensagem enviada com sucesso pelo DCB. Quando a confirmação é recebida, o módulo *Code* monta a próxima mensagem de atualização de atributo que está armazenada na primeira posição da fila de saída.

A implementação de mecanismos de sincronização e gerenciamento de propriedade não é objetivo desse trabalho, por isso esses aspectos são deixados de lado no protótipo. No entanto, para tornar viável a realização do estudo de caso, existe a real necessidade de garantir um mínimo de coordenação entre as entidades participantes.

Como pode ser visto na FIGURA 5.4, foi implementado de forma provisória um mecanismo para garantir a ordenação das mensagens, semelhante ao *handshake* descrito em [HAN98]. Esse mecanismo garante que uma nova mensagem somente será enviada depois que a anterior for recebida com sucesso. Enquanto as mensagens são recebidas do federado e não podem ser enviadas, ficam armazenadas na fila de saída do EDCB.

Uma visão mais detalhada dos módulos implementados no embaixador do DCB pode ser encontrada na listagem do ANEXO 4.

5.4 Embaixador do Federado (EF)

O embaixador do federado (FIGURA 5.5) é responsável pelo registro dos atributos de entrada (nome, tipo e propriedade) que compõem a interface do federado a que pertence (módulo *InputRegister*). Esses atributos podem ser constantes ou variáveis e são classificados em três tipos: IN, OUT ou IN/OUT.

O EF também é responsável pelo recebimento das mensagens provenientes do DCB, armazenamento na fila de atributos de entrada e notificação do *gateway* quanto à chegada de novos valores (módulo *Decode*). Mais detalhes sobre a implementação do EF podem ser obtidos na listagem do ANEXO 3.

O EF futuramente deverá tratar de aspectos de sincronização (ex. detecção de violação a LCC, mecanismo de *rollback*). Porém, neste trabalho, essas funcionalidades ainda estão em aberto e precisam ser adicionadas.

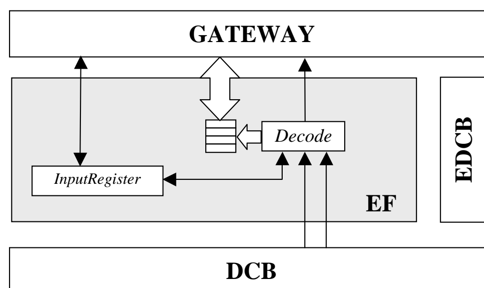


Figura 5.5: Protótipo do embaixador do federado (EF)

5.5 Templates do Gateway

Os *templates* têm o objetivo de facilitar a tarefa de construção da interface (*gateway*) responsável pela integração / interoperabilidade entre a infraestrutura de co-simulação e os federados. A idéia inicial, proposta em [MEL2001], previa que o *gateway* fosse construído como um dos módulos do embaixador do federado (EF). No entanto, com o desenvolvimento de pequenos protótipos, foi percebido que (1) era viável a construção de embaixadores estáticos (sem a necessidade de geração de código e posterior compilação) e configuráveis, e (2) que somente seria necessária a geração de parte do código responsável especificamente pela interface com o federado. Por isso, a melhor alternativa foi tornar o *gateway* uma camada independente do embaixador, de forma que somente o *gateway* precisa ser recompilado.

A implementação do *gateway* sofre influência do tipo do federado que será conectado a ele, por isso foram construídos diferentes *templates*, com o objetivo de atender as mais diversas necessidades. A utilização dessa abordagem permite que, com a evolução da arquitetura DCB, novos *templates* sejam criados e módulos para configuração desses *templates* sejam adicionados na ferramenta de configuração (descrita na Seção 5.6), que auxilia no trabalho de configuração dos *templates*.

```
public class Gateway {
   // INICIALIZAÇÃO DAS VARIÁVEIS DE CLASSE...
   .
   .
   public Gateway(ApplicationDCB pointer) {
      // PROCEDIMENTOS PARA INICIALIZAÇÃO DO FEDERADO...
   }

   public void ProtocolConverter(int AttributeID) {
      // CÓDIGO GERADO PELA FERRAMENTA DE CONFIGURAÇÃO
      // ESSE PROCEDIMENTO RECEBE O ID DO ATRIBUTO
      // ARMAZENADO EM UMA FILA, BUSCA SEU VALOR
      // E O ENVIA PARA A INTERFACE DO FEDERADO.
      // NO CASO DE CHAMADAS DE FUNÇÕES, O PROCEDIMENTO
      // AGUARDA PELA CHEGADA DE TODOS OS ATRIBUTOS
      // UTILIZADOS PARA EXECUTAR A CHAMADA.
   }

   // DECLARAÇÃO DOS PROCEDIMENTOS DA INTERFACE DO FEDERADO
   // ESSES PROCEDIMENTOS PERMITEM QUE O GATEWAY ENVIE
   // DADOS PARA O FEDERADO

   .
   .
   // PROCEDIMENTOS PARA CONVERSÃO DE DADOS
   public int ToInt(String value) {
      return Integer.parseInt(value);
   }
   .
   .
   // FUNÇÕES SOBRECARREGADAS PARA RECEBER DADOS DO FEDERADO
   public void UpdateAttribute(String Name, String Value) {
      App.DCBA.Update(Name,String.valueOf(Value));
   }
   .
```

Figura 5.6: Trecho de código do *template* Java

A construção de um novo *template* para a arquitetura DCB é uma tarefa manual e exige conhecimentos detalhados da nova linguagem ou simulador que está sendo disponibilizado. No entanto, depois de construído o *template*, o projetista pode utilizá-lo para realizar a geração de *gateways* para diferentes modelos da linguagem/simulador a partir dos recursos disponibilizados pela ferramenta de configuração.

A especificação dos *templates* para a arquitetura DCB foi feita a partir da identificação das necessidades das diferentes formas de comunicação. Essas necessidades estão diretamente relacionadas ao tipo de interface do federado que está sendo adicionado à federação. Até o presente momento, é possível a integração de federados, utilizando *templates*, nos seguintes contextos:

1. Federados localizados na mesma máquina da infraestrutura DCB – nesse contexto os federados podem ser integrados das seguintes formas:

    a) Federados que possuem interfaces Java - são integrados de forma direta e a comunicação é realizada através de chamadas de funções e passagem de parâmetros. É possível a comunicação tanto com federados que possuem código fonte quanto compilados (.class);

    b) Federados que possuem interfaces C/C++ - podem ser integrados de três diferentes maneiras:

        a. Quando implementados como bibliotecas dinâmicas (.dll ou .so) ou que oferecem sua interface por meio de uma delas – são carregados diretamente pelo *gateway* e a comunicação é feita através de rotinas de acesso a código nativo oferecidas pela JNI (*Java Native Interface*). Essa abordagem também torna possível a criação de *wrappers* para a interface do federado;

        b. Quando possuem código fonte disponível – podem ser integrados através da agregação de um *template* ao seu código original. Esse *template* invoca a JVM e torna possível realizar chamadas de funções (através da *Java Native Interface* - JNI) aos objetos Java que compõem a infraestrutura de co-simulação;

        c. Quando implementados como código objeto (ex. arquivos .obj) podem ser carregados através de um *wrapper* customizado que pode ser uma biblioteca ou um executável. Esse *wrapper* oferece uma interface acessível à infraestrutura de co-simulação;

2. Federado localizado em um *host* remoto, ou seja, localizado em uma máquina diferente da infraestrutura de co-simulação – podem ser integrados através de mecanismos de comunicação interprocessos como *sockets*;

3. Federados que não possuem nenhum tipo de mecanismo para comunicação direta – podem ser integrados de duas formas alternativas (ainda não implementadas):

a) Através de leitura ou escrita em arquivos pré-definidos;

b) Através de mecanismos de leitura e escrita em posições de memória compartilhada.

Esses diferentes métodos de interoperabilidade são implementados em *templates* distintos com o objetivo de facilitar a integração do maior número de federados existentes. As funcionalidades e requisitos dos *templates* descritos nos capítulos seguintes são resultado de diversos protótipos implementados no decorrer desse trabalho.

Em um próximo passo do trabalho, tomando como base os *templates* já existentes, é possível a construção de modelos mais elaborados que permitam a integração automática ou semi-automática de módulos descritos em linguagens de descrição de hardware como VHDL e SystemC.

5.5.1 *Template* Java

O *template* Java, visualizado na FIGURA 5.7, é o *template* básico oferecido pela arquitetura DCB. É utilizado nos casos em que o federado foi escrito em Java e possui uma interface definida, o que permite a instanciação do objeto federado e a troca de informações através de chamadas de métodos. Esse método é forma mais direta de integração e pode ser classificado dentro do item (a) da FIGURA 4.10.

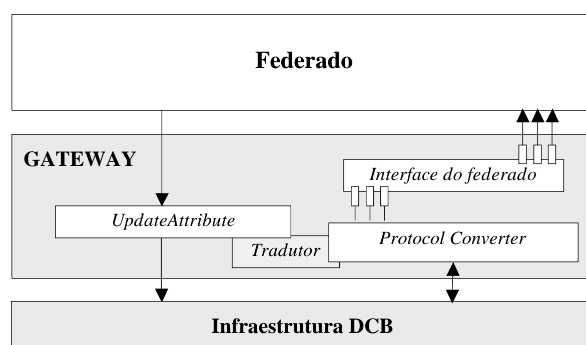


Figura 5.7: Detalhes internos do *template* Java

O *gateway* para integrar federados descritos em Java pode ser facilmente gerado a partir da especificação fornecida pela ferramenta de modelagem. O código do *template* desenvolvido para esse fim pode ser encontrado no ANEXO 1. As principais funcionalidades dos módulos que fazem parte do *template* Java são as seguintes:

1. Conversor de protocolos (*ProtocolConverter*) – módulo utilizado para recebimento de novos valores de atributos e para coordenação das conversões. Tem a responsabilidade de identificar um atributo ou conjunto de atributos recebidos e realizar as conversões necessárias. Uma das possibilidades, visualizada na FIGURA 5.8, é a chamada de funções do

federado. Esse tipo de funcionalidade é necessária, pois a arquitetura DCB trabalha somente com envio e recebimento de conjuntos atributo-valor.

2. *UpdateAttribute* – responsável pelo recebimento de atualizações de valores de atributos do federado. Para que os valores sejam atualizados, é necessário que o federado chame esse procedimento.

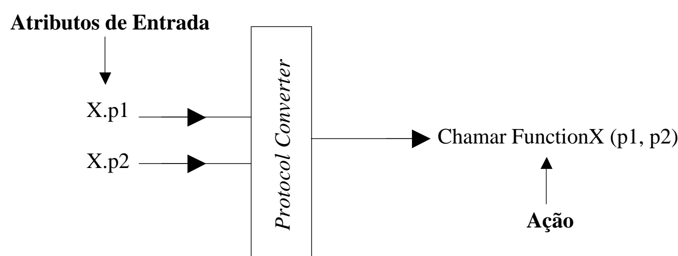


Figura 5.8: Detalhes do funcionamento do conversor de protocolos

3. Tradutor – é um conjunto implícito de funções responsáveis pela tradução de tipos de dados. Essas funções são utilizadas pelo conversor de protocolos (*ProtocolConverter*) e pelo módulo *UpdateAttribute*.

5.5.2 Template Sockets

O *template sockets* foi desenvolvido para resolver problemas de comunicação com federados que não possuem uma interface para acesso direto ou federados que estão localizados em *hosts* remotos. Assim, a comunicação entre o federado e o *gateway* pode ser classificada como semidireta, conforme descrito no item (b) da FIGURA 4.10.

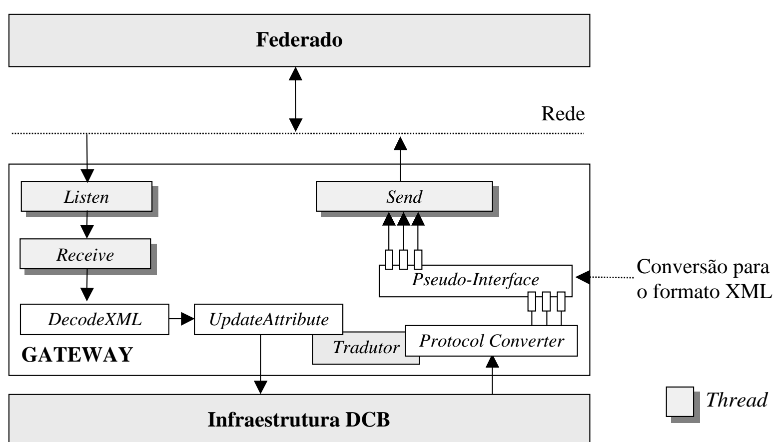
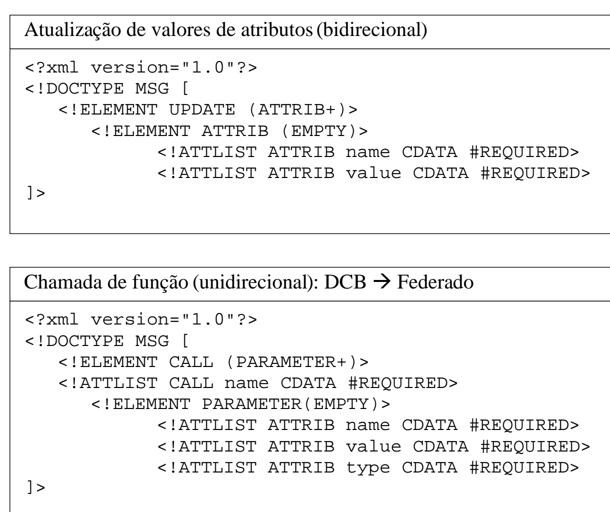


Figura 5.9: Detalhes internos do *template sockets*

Esse *template*, detalhado na FIGURA 5.9, oferece uma interface padronizada para troca de mensagens através de *sockets* TCP. Além dos módulos *UpdateAttribute*, *ProtocolConverter* e Tradutor (descritos em 5.5.1) do *template* Java, ele possui algumas funcionalidades adicionais, descritas a seguir:

1. Pseudo-Interface – esse módulo possui a declaração da interface real do federado que está localizado no *host* remoto. Oferece funções para a conversão de atualizações de atributos ou chamadas de função em mensagens codificadas em XML, que são enviadas pela rede. Essas mensagens seguem o padrão descrito na FIGURA 5.10.
2. *Listen*, *Send* e *Receive* – procedimentos implementados utilizando *sockets* Java que executam respectivamente o recebimento de nova conexão, tratamento da conexão e envio de dados pela rede.
3. *DecodeXML* – oferece rotinas específicas para decodificação de mensagens recebidas do federado. Esse módulo faz a decodificação das informações e atualiza os atributos necessários. As mensagens recebidas também devem seguir o formato especificado na FIGURA 5.10.



Atualização de valores de atributos (bidirecional)

```
<?xml version="1.0"?>
<!DOCTYPE MSG [
   <!ELEMENT UPDATE (ATTRIB+)>
      <!ELEMENT ATTRIB (EMPTY)>
            <!ATTLIST ATTRIB name CDATA #REQUIRED>
            <!ATTLIST ATTRIB value CDATA #REQUIRED>
]>
```

Chamada de função (unidirecional): DCB → Federado

```
<?xml version="1.0"?>
<!DOCTYPE MSG [
   <!ELEMENT CALL (PARAMETER+)>
   <!ATTLIST CALL name CDATA #REQUIRED>
      <!ELEMENT PARAMETER(EMPTY)>
            <!ATTLIST ATTRIB name CDATA #REQUIRED>
            <!ATTLIST ATTRIB value CDATA #REQUIRED>
            <!ATTLIST ATTRIB type CDATA #REQUIRED>
]>
```

Figura 5.10: Especificação formal (DTD) das mensagens trocadas entre federado e *gateway*

A utilização de comunicação via *sockets* em conjunto com a formatação de mensagens em XML no *template sockets* faz com que a conexão do *gateway* com o federado seja feita de forma independente de plataforma e de linguagem. No entanto, essa abordagem exige muitos requisitos específicos no lado do federado. Esses requisitos, conforme ilustrado na FIGURA 5.11, são basicamente os mesmos implementados no *template sockets* (FIGURA 5.9), o que, em muitos casos, poderá inviabilizar a integração através deste método.

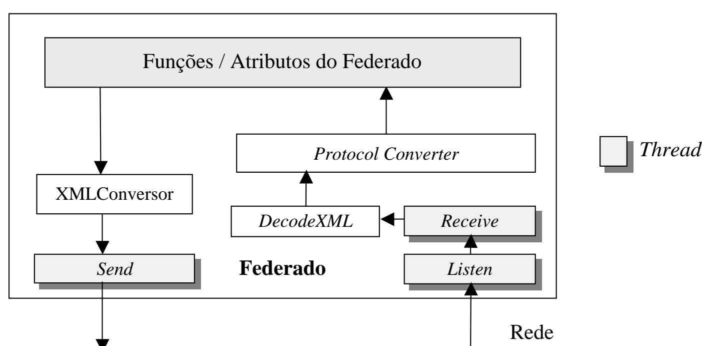


Figura 5.11: Requisitos para um federado comunicar-se através de *sockets*

5.5.3 *Template* JNI

O *template* JNI é o mais complexo, pois permite que o *gateway* comunique-se com código nativo, escrito em linguagens como C/C++. Esse *template*, que pode ser classificado dentro do item (c) da FIGURA 4.10, permite a integração de federados através do uso de uma funcionalidade específica da plataforma Java, a JNI (*Java Native Interface*) que permite que objetos Java executem e recebam chamadas de código nativo. A geração do código desse *gateway* pode ocorrer de forma semi-automática e, em muitos casos, exigirá a intervenção do usuário para que a integração efetiva seja realizada.

Foram identificadas duas abordagens diferentes para comunicação com código nativo. Na primeira o *template* do *gateway* carrega uma biblioteca (*.dll no windows ou .so no linux*) que serve de *wrapper* para a interface do federado. Na segunda abordagem, é disponibilizado ao federado (ex. código fonte ou código objeto) um *template* que invoca a JVM, inicializa a infraestrutura DCB e registra os métodos nativos deste federado, permitindo a comunicação bi-direcional. A seguir as duas abordagens são descritas de forma mais detalhada.

5.5.3.1 Integração através de uma biblioteca de ligação dinâmica

Nesse caso de integração, é levado em conta que o federado está implementado dentro de uma biblioteca de ligação dinâmica (ex. uma .dll ou .so) ou compilado em código objeto (ex. .obj). O *template* JNI é responsável por acessar as funcionalidades do federado através de um *wrapper* possibilitando a comunicação síncrona entre o objeto Java (*gateway*) e o código nativo (federado). No *template* que carrega uma biblioteca dinâmica, o *gateway* é responsável por declarar os métodos nativos do federado e por carregar a biblioteca chamando a primitiva "System.LoadLibrary".

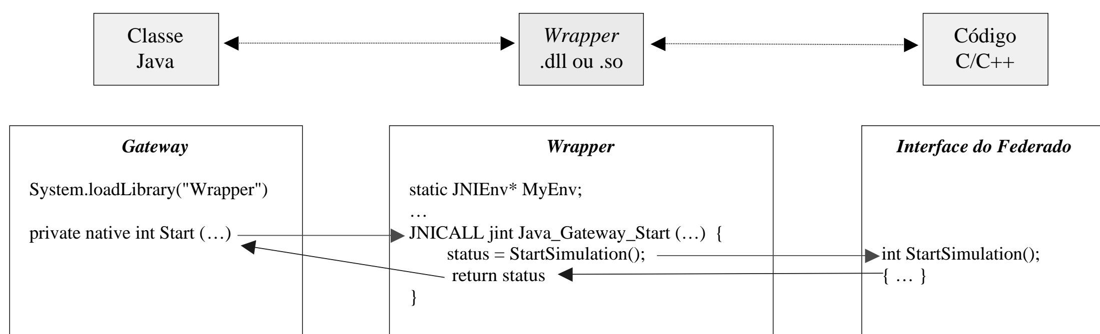


Figura 5.12: Exemplo de funcionamento da JNI (*Java Native Interface*)

A FIGURA 5.12 exemplifica o funcionamento da comunicação síncrona oferecida pela JNI. A comunicação síncrona JNI permite que objetos Java executem chamadas a métodos nativos e que código nativo faça chamadas a objetos Java (*callback*). No entanto, as chamadas *callback* somente são executadas com sucesso dentro da "*thread*" que está sendo executada, ou seja, no momento em que o fluxo de execução do método nativo termina, a referência à JVM não é mais válida. Assim, se o federado possuir *threads* nativas, em alguns determinados casos, a chamada *callback* não será executada.

Para resolver esse problema, a JNI permite que seja feito um "*attach*" da variável de ambiente da JVM (JNIEnv) à *thread* corrente. No entanto, em alguns casos não é possível alterar determinado segmento de código do federado para inserir essas chamadas, inviabilizando a comunicação.

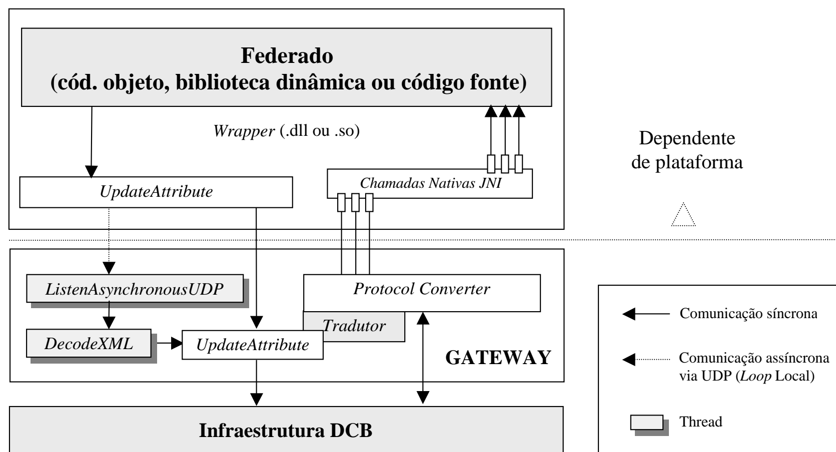


Figura 5.13: Detalhes internos do *template* JNI (*Java Native Interface*)

Como em algumas situações a utilização de chamadas assíncronas é indispensável, torna-se necessária a implementação de um método alternativo para resolver o problema. Para isso, são implementados no *gateway* e no *wrapper* módulos de comunicação que utilizam *sockets* UDP no *loop* local (FIGURA 5.13). A definição do modo de comunicação é automática (FIGURA 5.14) e não necessita intervenção do usuário.

Funcionalmente, o *gateway*, através da biblioteca *wrapper*, faz as chamadas ao federado. Se necessário, o federado (durante a execução), faz chamadas ao método *UpdateAttribute* para atualização de atributos. Caso seja necessário algum tipo de comunicação assíncrona (ex. quando o ponteiro para JVM não é mais válido), o método *UpdateAttribute* resolve internamente a forma de envio do valor do atributo atualizado para o federado, conforme detalhado na FIGURA 5.14.

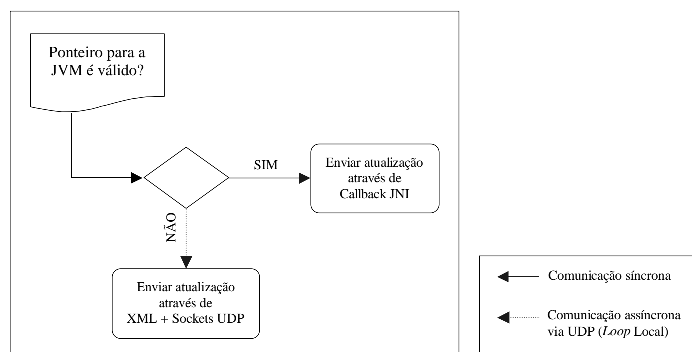


Figura 5.14: Esquema de tomada de decisão da função *UpdateAttribute*

5.5.3.2 Integração através de chamada direta a JVM (Java Virtual Machine)

Essa abordagem é uma alternativa desenvolvida para simplificar a integração e eliminar o uso da biblioteca dinâmica responsável pelo carregamento do federado. A FIGURA 5.15 detalha essa técnica de integração, onde é oferecido ao federado um *template* que realiza três principais tarefas: a instanciação direta da JVM, a inicialização da infraestrutura DCB e o registro dos métodos nativos do federado junto ao objeto *gateway*. O código desse *template* está parcialmente listado no ANEXO 2.

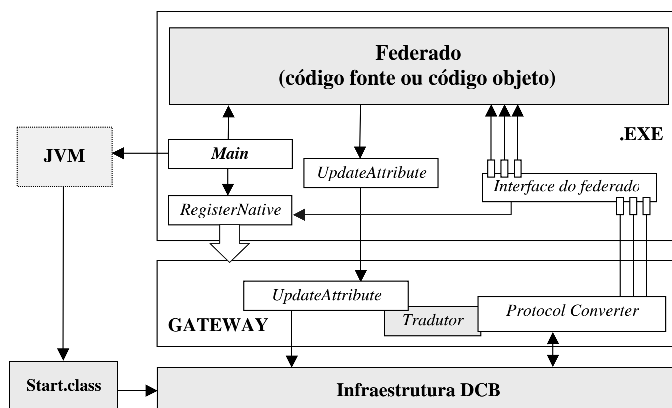


Figura 5.15: Abordagem alternativa para integração de federados

A utilização do *template* JNI torna possível a comunicação bi-direcional entre o federado e a infraestrutura de co-simulação sem a necessidade de adaptações alternativas para permitir a comunicação assíncrona, como a utilizada no *template* que carrega a biblioteca *wrapper* (Capítulo 5.5.3.1). Essa abordagem permite a integração de federados que possuem ou não código fonte.

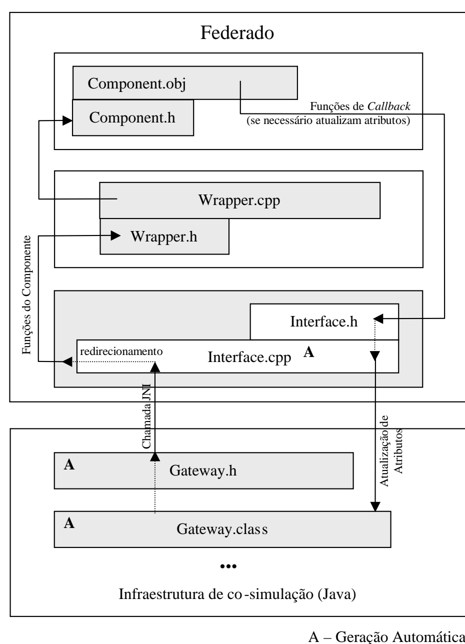


Figura 5.16: Exemplo de integração de um federado C/C++

A FIGURA 5.16 ilustra um exemplo da integração de um federado escrito em C++ e compilado como código objeto. Para efetivar a integração é necessário que o *gateway* possua um arquivo *header* (gateway.h) que contenha a declaração da interface do federado. Quando existir a necessidade de atualizações de valores de atributos durante a execução do federado, também é necessário que uma interface de *callback* para o federado seja declarada em um arquivo *header* (interface.h). Essa interface é implementada no módulo Interface.cpp e redireciona os valores de atributos para a classe *gateway* através de chamadas JNI. Opcionalmente é possível utilizar uma camada *wrapper*, em casos onde exista a necessidade de alterar a forma de acesso ou o protocolo utilizado pela interface do federado.

5.6 Ferramenta para configuração dos embaixadores e geração do *gateway*

A ferramenta de configuração, descrita na FIGURA 5.17, foi desenvolvida utilizando a linguagem Java. Seu principal objetivo é facilitar o processo de implementação da co-simulação exigindo o mínimo possível de intervenção do usuário. A ferramenta possui duas principais tarefas: a geração dos *gateways* com base em *templates* pré-definidos e a geração dos arquivos XML para configuração dos embaixadores.

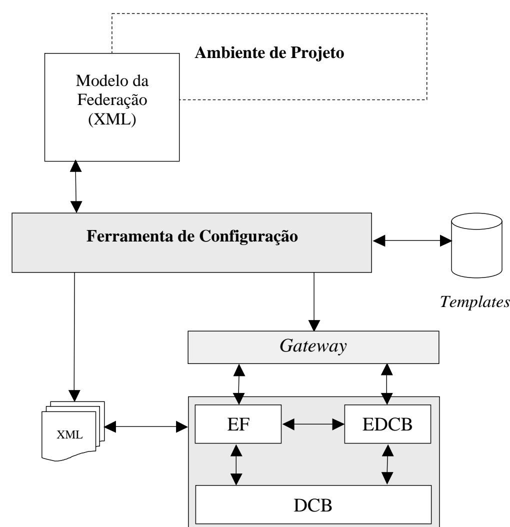


Figura 5.17: Ferramenta de configuração

As informações necessárias para que a ferramenta de configuração execute suas tarefas são disponibilizadas pelo projetista através de um arquivo XML, chamado de modelo da federação, que é especificado formalmente na FIGURA 5.18. A geração do modelo da federação é feita com auxílio de uma ferramenta visual (fruto de outra dissertação de mestrado) que permite a adaptação e a integração de componentes (federados). O modelo gerado por essa ferramenta contém informações sobre os federados, descrição de suas interfaces e ligações entre atributos das interfaces, como pode ser visto com mais detalhes no exemplo listado no ANEXO 6.

```
<?xml version="1.0"?>
<!DOCTYPE config [
   <!ELEMENT federation (federate+)>
      <!ATTLIST federation id CDATA #REQUIRED>
      <!ELEMENT federate (attribute*, function*)>
          <!ATTLIST federate id CDATA #REQUIRED>
          <!ATTLIST federate ip CDATA #REQUIRED>
          <!ATTLIST federate port CDATA #REQUIRED>
          <!ATTLIST federate type CDATA #REQUIRED>
          <!ATTLIST federate code_location CDATA #REQUIRED>
          <!ATTLIST federate interface_tipe CDATA #REQUIRED>

          <!ELEMENT attribute (destination+)>
             <!ATTLIST attribute uid CDATA #REQUIRED>
             <!ATTLIST attribute name CDATA #REQUIRED>
             <!ATTLIST attribute type CDATA #REQUIRED>
             <!ATTLIST attribute port_type (IN|OUT|INOUT) "INOUT">
             <!ELEMENT destination (EMPTY)>
                <!ATTLIST destination federationid CDATA #REQUIRED>
                <!ATTLIST destination federateid CDATA #REQUIRED>
                <!ATTLIST destination attribute CDATA #REQUIRED>

          <!ELEMENT function (parameter*,return*)>
             <!ATTLIST attribute uid CDATA #REQUIRED>
             <!ATTLIST attribute name CDATA #REQUIRED>
             <!ATTLIST attribute return_type CDATA #REQUIRED>
             <!ELEMENT return (destination+)>
             <!ELEMENT parameter (EMPTY)>
                <!ATTLIST parameter uid CDATA #REQUIRED>
                <!ATTLIST parameter name CDATA #REQUIRED>
                <!ATTLIST parameter type CDATA #REQUIRED>
]>
```

Figura 5.18: Especificação formal (DTD) do arquivo gerado pela ferramenta de modelagem

A ferramenta de configuração, visualizada com mais detalhes na FIGURA 5.19, a partir das informações extraídas do modelo da federação, cria uma estrutura de objetos Java que armazena todas as informações contidas no modelo da federação. Com base nessas informações a ferramenta gera, para cada federado do modelo, a configuração dos embaixadores e o código do *gateway*.

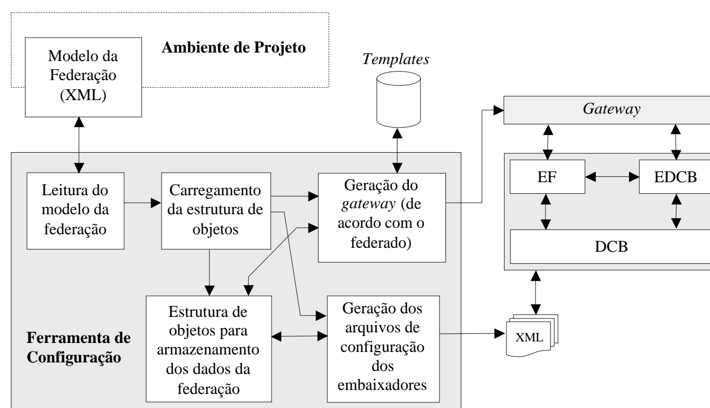


Figura 5.19: Detalhes internos da arquitetura da ferramenta de configuração

O módulo de configuração dos embaixadores gera de forma automática, para cada um dos federados participantes da simulação, o arquivo XML (especificado formalmente na FIGURA 5.20) que é utilizado pelos embaixadores. Esse arquivo é lido no momento da inicialização da infraestrutura de co-simulação, o que evita a recompilação do código dos embaixadores. Como exemplo, pode ser encontrada no

ANEXO 5 a listagem do arquivo de configuração utilizado em um dos experimentos do estudo de caso.

```
<?xml version="1.0"?>
<!DOCTYPE federateconfig [
   <!ELEMENT federateinfo (attribute)>
      <!ATTLIST federateinfo federateid CDATA #REQUIRED>
      <!ATTLIST federateinfo federationid CDATA #REQUIRED>
      <!ATTLIST federateinfo localport CDATA #REQUIRED>
      <!ATTLIST federateinfo type (synchronous|assynchronous) "synchronous">
      <!ELEMENT attribute (EMPTY)>
              <!ATTLIST attribute id CDATA #REQUIRED>
              <!ATTLIST attribute name CDATA #REQUIRED>
              <!ATTLIST attribute type CDATA #REQUIRED>

   <!ELEMENT federation (federate)>
      <!ELEMENT federate (EMPTY)>
          <!ATTLIST federate id CDATA #REQUIRED>
          <!ATTLIST federate ip CDATA #REQUIRED>
          <!ATTLIST federate port CDATA #REQUIRED>

   <!ELEMENT outputattribute (destination)>
      <!ELEMENT destination (EMPTY)>
          <!ATTLIST federate federationid CDATA #REQUIRED>
          <!ATTLIST federate federateid CDATA #REQUIRED>
          <!ATTLIST federate attribute CDATA #REQUIRED>
]>
```

Figura 5.20: Especificação formal (DTD) do arquivo de configuração dos embaixadores

O módulo de geração do *gateway* é responsável pela identificação do tipo de *gateway* que deverá ser gerado para cada federado participante da co-simulação. Isso é determinado de acordo com as especificações de interface fornecidas pelo projetista no modelo da federação. Depois disso, pode ser gerado um *gateway* para cada um dos participantes da simulação (com base nos *templates* descritos no Capítulo 5.5). Esta tarefa, por ser baseada na geração de código a partir de um *template* pré-definido, exige a recompilação do código do *gateway*.

A tarefa de geração do código do *gateway*, em alguns casos, pode exigir a intervenção manual do projetista (semi-automática) para completar a integração. Isso ocorre porque, dependendo do *template* utilizado, nem todos os tipos e formatos de interfaces existentes ou situações de integração podem ser previstas.

6 ESTUDO DE CASO

Como estudo de caso, foi utilizado o modelo GPSAlerta [LIS2002] desenvolvido no PPGC pelo Prof. Carlos Arthur Lang Lisboa. O GPSAlerta foi modelado para ser conectado a um dispositivo GPS (*Global Position System*) para monitorar a posição geográfica do veículo no qual está instalado e emitir alertas ao usuário sobre a proximidade de pontos geográficos previamente cadastrados.

Funcionalmente, o terminal GPSAlerta recebe coordenadas, compara-as com os pontos definidos pelo usuário (armazenados na memória do sistema), e exibe alertas ao usuário sobre a proximidade de pontos previamente armazenados. O terminal foi projetado para ser conectado a um Garmin GPS 31/31 SL TrackPak por meio de uma interface RS-232. A comunicação entre o GPS e o terminal ocorre através de mensagens ASCII. O objetivo final do trabalho é que o GPSAlerta seja sintetizado em uma placa (FPGA) e futuramente incorporado como parte de um celular.

Para dar seguimento aos experimentos planejados para este trabalho, o modelo GPSAlerta sofreu algumas alterações, com o objetivo de torná-lo mais adequado para experimentação e validação dos protótipos desenvolvidos. O processo de validação dos protótipos dos embaixadores, *gateway* e da ferramenta de modelagem foi baseado na modelagem, configuração e execução de diferentes modelos de co-simulação. Esses diferentes modelos foram construídos a partir de diferentes configurações do GPSAlerta, através da adição ou exclusão de componentes e na alternância entre diferentes níveis de abstração.

A metodologia utilizada para executar a co-simulação de cada um dos modelos pode ser dividida em 3 etapas:

- A geração do modelo simulável (federação) é iniciada na ferramenta visual de modelagem, onde é descrita a interface de cada um dos componentes do modelo;
- Em um segundo passo são definidas as interações que ocorrerão entre os atributos da interface dos componentes. Finalizada essa etapa, a ferramenta de modelagem disponibiliza um arquivo XML que contém o modelo da federação;
- Utilizando como base o modelo da federação, a ferramenta de configuração (descrita na Seção 5.6) faz a geração automática de: (a) um *gateway* para cada um dos componentes do modelo e (b) dos arquivos de configuração utilizados pelos embaixadores (um arquivo XML para cada par de embaixadores).

Os protótipos desenvolvidos foram construídos em Java e C++. A geração dos *gateways* e do arquivo XML utilizado para configuração dos embaixadores desse

modelo foi feita de forma automática com auxílio da ferramenta de configuração. Apesar da geração ser considerada automática, em alguns casos foram necessárias pequenas intervenções, como por exemplo, na inserção de algumas linhas de código para a inicialização de um federado ou a conversão de um valor de atributo para um tipo ainda não previsto pelo módulo tradutor do *gateway* (ex. hexadecimal).

Os experimentos realizados têm como principal finalidade a demonstração das funcionalidades dos protótipos desenvolvidos e também do potencial dos recursos oferecidos pela infraestrutura DCB. A seguir são detalhados cada um dos modelos utilizados nos experimentos.

6.1 Experimento 1

Nesse primeiro experimento, que pode ser chamado de modelo comportamental (*behavioral*), tanto a computação quanto a comunicação são descritos em um alto nível de abstração. O modelo GPS Alerta é composto por quatro componentes que interagem através de primitivas de comunicação de alto nível (ex. *send*, *receive*). Essas primitivas podem possuir parâmetros, no entanto nesse nível os parâmetros utilizados são de um tipo pré-definido, que neste caso é o *string*.

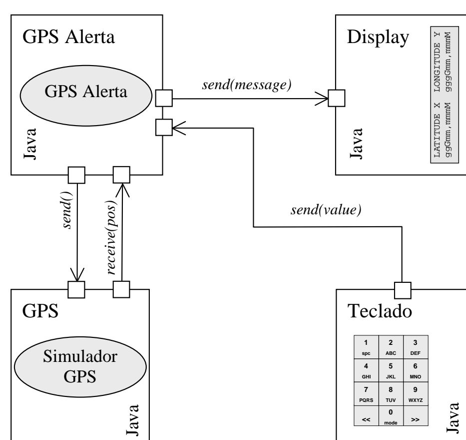


Figura 6.1: GPS Alerta modelado em um alto-nível de abstração

No modelo comportamental, que pode ser visualizado na FIGURA 6.1, ainda não existem informações sobre a arquitetura do sistema. A computação dos componentes é modelada em uma linguagem de alto nível (Java), dividida entre quatro tarefas (federados) distintas, conforme descrito abaixo:

1. GPS Alerta – é o módulo principal do sistema. Tem como principais tarefas solicitar e receber dados do GPS, calcular e comparar posições geográficas, atender a chamadas do teclado e enviar dados para exibição no *display*;
2. Simulador GPS – simula a geração de uma seqüência de coordenadas, da mesma forma como um GPS real. Essa aplicação, modelada em Java, lê

coordenadas de um arquivo texto e as disponibiliza para o GPS Alerta, de acordo com a formatação descrita em [LIS2002];

3. Teclado – aplicação Java que simula um teclado numérico. O pressionamento de uma tecla implica na geração de um valor de saída, que é enviado para o GPS Alerta;
4. *Display* – aplicação Java que simula um *display* gráfico de 2 linhas x 24 caracteres. É utilizada somente para fins de simulação, assim como o teclado e o Simulador GPS.

Esse primeiro experimento é baseado em um modelo extremamente simples, cujo principal objetivo é oferecer ao projetista a possibilidade de fazer uma avaliação inicial do comportamento do modelo GPS Alerta. O modelo apresentado neste experimento, após sofrer um processo de refinamento, servirá como base para o modelo utilizado no experimento seguinte. Essa abordagem, baseada no processo de evolução progressiva do modelo de co-simulação, auxilia o projetista no processo de escolha dos componentes mais adequados para o seu sistema. Isso também influi na complexidade da validação dos protocolos de comunicação, que pode ser dividida em etapas, utilizando diferentes níveis de detalhamento.

A TABELA 6.1 apresenta mais informações sobre o primeiro experimento executado.

Tabela 6.1: Detalhes do experimento 1

Federado	Linguagem do federado	Tamanho do federado (linhas)	Tamanho do *gateway* gerado (linhas)	Template utilizado	Tamanho do XML gerado para config. do EF e EDCB (linhas)
GPS Alerta	Java	348	158	Java	17
Simulador GPS	Java	107	127	Java	12
Display	Java	439	145	Java	8
Teclado	Java	230	130	Java	12

6.2 Experimento 2

No segundo experimento previsto para o estudo de caso, o modelo já reflete uma arquitetura abstrata do sistema. O GPS Alerta não é mais considerado um simples componente do sistema, mas sim uma tarefa executada pelo micro-controlador escolhido. Para esse estudo de caso foi utilizado o micro-controlador FemtoJava [ITO99] [ITO2000] [ITO2001], que pode ser visualizado na FIGURA 6.2.

No nível de abstração utilizado no segundo experimento, o FemtoJava ainda não é o componente real, mas sim um modelo comportamental descrito em Java. O modelo de comunicação disponibilizado pelo FemtoJava permite que a interação com os componentes ocorra através de dois principais meios: portas de entrada e saída (E/S) ou interrupções. As portas de E/S são utilizadas respectivamente para leitura e escrita de valores, já as chamadas de interrupção são utilizadas para interromper a execução do micro-controlador e posteriormente executar um determinado procedimento.

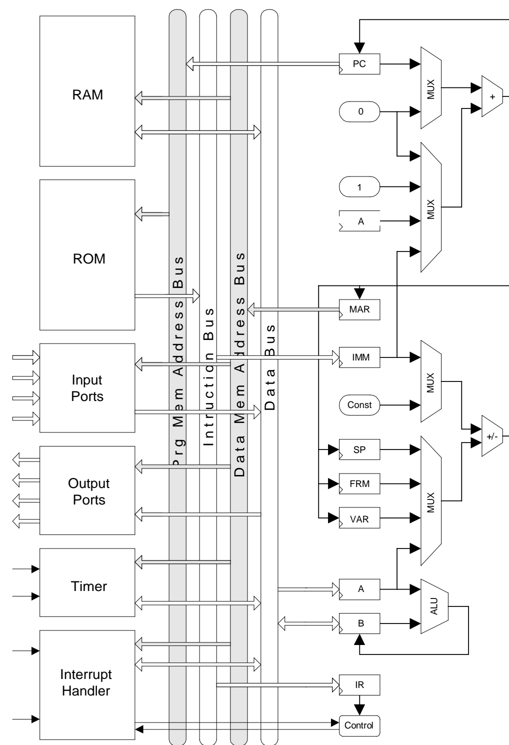


Figura 6.2: Detalhes do micro-controlador FemtoJava (Fonte: [ITO2001])

No modelo do nível 2 a comunicação sofre um processo de refinamento, passando do simples protocolo de mensagens (*send-receive*) para um protocolo de mais baixo nível, utilizando inclusive tipos de dados complexos (ex. *float, int, string*). Apesar da disponibilidade dos recursos do micro-controlador, estes ainda não são completamente utilizados nesse modelo. Ainda assim, mesmo com o emprego de um alto nível de abstração, é possível que sejam executados experimentos um pouco mais detalhados (em relação ao experimento anterior) para validar o comportamento dos componentes do GPS Alerta e dos protocolos de comunicação utilizados por eles.

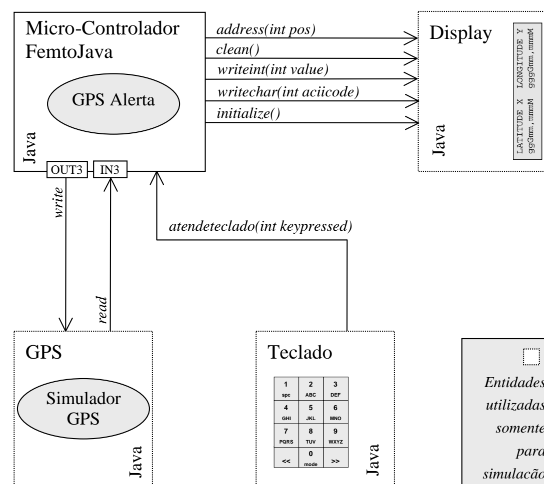


Figura 6.3: Modelo do GPSAlerta utilizado no experimento 2

Nesse segundo nível, o experimento é caracterizado pelo modelo de comunicação multi-nível, ou seja, em um mesmo modelo a comunicação intercomponentes é realizada em diferentes níveis de abstração. Como pode ser visto na FIGURA 6.3, são utilizadas as portas IN3 e OUT3 do FemtoJava para realizar a comunicação com o Simulador GPS e, ao mesmo tempo, são utilizadas primitivas de alto-nível para comunicação com os demais componentes do sistema.

A capacidade de agregar componentes com interfaces modeladas em diferentes níveis de abstração em um mesmo modelo de simulação permite que algumas partes de um determinado modelo sejam representadas em baixo nível enquanto que, outras partes são representadas em alto-nível de abstração. Isso permite que um experimento de co-simulação seja focado especificamente na validação da interação entre um determinado conjunto de componentes, o que poderia minimizar as perdas de performance ocasionadas pela comunicação entre componentes de menor importância parao sistema.

Em comparação com o modelo comportamental (descrito no experimento anterior), os demais componentes do sistema (GPS, *display* e teclado) não possuem alterações significativas na modelagem de seu comportamento. As principais alterações estão localizadas em suas interfaces, que sofreram um primeiro processo de refinamento no protocolo de comunicação, conforme detalhado abaixo:

- Simulador GPS – agora utiliza portas de entrada e saída para comunicar-se com o FemtoJava. Recebe uma solicitação através da porta OUT3 e disponibiliza um novo dado para leitura na porta IN3;
- *Display* – o conjunto de funções da interface desse componente foi mais bem detalhado. Foram adicionadas funcionalidades responsáveis pela inicialização do display, limpeza da tela, posicionamento do cursor e escrita de dados. Essas funcionalidades deverão ser executadas futuramente pelo *driver* de software do *display*;
- Teclado – sofreu um pequeno refinamento na comunicação, passando a utilizar um tipo de dado específico (*int*) no evento de notificação do GPS Alerta. Assim como o *display*, o teclado continua sendo utilizado somente para fins de simulação.

A TABELA 6.2 apresenta mais informações sobre o experimento realizado. Nota-se que, comparados ao experimento anterior, alguns dos *gateways* e arquivos de configuração sofrem um pequeno acréscimo nos seus tamanhos, como conseqüência do detalhamento da interface de comunicação. Além disso, a necessidade de adaptação das interfaces de alguns componentes também contribuiu para o aumento no tamanho dos mesmos, como foi o caso do *Display*.

Tabela 6.2: Detalhes do experimento 2

Federado	Linguagem do federado	Tamanho do federado (linhas)	Tamanho do *gateway* gerado (linhas)	Template utilizado	Tamanho do XML gerado para config. do EF e EDCB (linhas)
GPS Alerta	Java	350	158	Java	27
Simulador GPS	Java	107	127	Java	12
Display	Java	600	203	Java	11
Teclado	Java	230	130	Java	12

6.3 Experimento 3

O modelo de co-simulação utilizado no experimento 3, ilustrado na FIGURA 6.4, já reflete a arquitetura final do sistema, mesmo que ainda não esteja modelado no mais baixo nível de abstração possível (RTL – *Register Transfer Level*). Este último experimento é caracterizado pela computação multi-nível, em que componentes modelados em diferentes níveis de abstração cooperam de forma transparente.

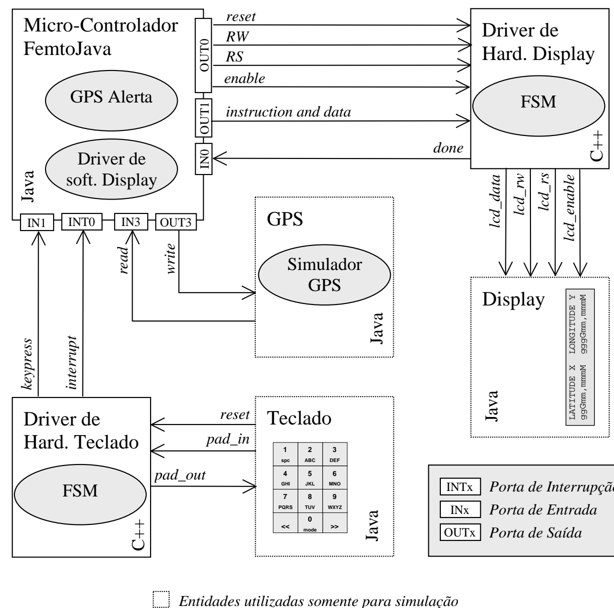


Figura 6.4: Modelo GPSAlerta utilizado no experimento 3

Os maiores diferenciais desse modelo com relação ao utilizado no experimento anterior são: a adição de novos componentes ao sistema (*driver* de hardware/software do *display* e *driver* de hardware do teclado) e o novo processo de refinamento sofrido pelas interfaces de comunicação. Essas modificações são detalhadas a seguir:

b. *Driver* de hardware do *display* – foi adicionado ao modelo para executar as tarefas relacionadas ao gerenciamento do hardware do *display* real. É baseado em uma máquina de estados finita (*Finite State Machine* - FSM) originalmente descrita em VHDL. Esse modelo VHDL [SAS2002] foi obtido junto aos códigos que compõem o FemtoJava original e foi reescrito em C++. A comunicação com o *gateway* é feita através de uma interface JNI (*Java Native Interface*);

c. *Driver* de hardware do teclado – foi inserido no modelo para controlar o hardware do teclado real. Assim como o *driver* do *display*, esse componente é baseado em máquina de estados finita (FSM). O *driver* do teclado é responsável pela varredura e identificação da tecla pressionada pelo usuário, notificação do GPS Alerta através de uma chamada a interrupção INT0 do FemtoJava e posterior escrita do valor da tecla pressionada na porta IN1. A máquina de estados implementada nesse modelo foi construída a partir de um

modelo VHDL já existente e reescrita em C++, utilizando uma interface JNI para se comunicar com a infraestrutura de co-simulação;

d. *Driver* de software do *display* – foi introduzido no modelo com o objetivo de disponibilizar ao GPS Alerta funções de alto nível para controle do *display*. A comunicação dessa tarefa com o *driver* de hardware do *display* é realizada através das portas de saída OUT0 e OUT1 do FemtoJava;

e. *Display* – esse componente sofreu um refinamento em sua interface de comunicação. O conjunto de sinais de entrada foi alterado para prover uma simulação mais realista do modelo e para torná-lo compatível com o *driver* de hardware utilizado.

f. Teclado – Assim como ocorreu com o *display*, a interface de comunicação do teclado também sofreu um processo de refinamento, oferecendo recursos para o *driver* do teclado executar o processo de varredura, que resulta na identificação da posição (linha/coluna) da tecla pressionada.

Por ser o modelo simulável mais próximo do sistema real, pode-se verificar que a comunicação entre os componentes já reproduz de forma fiel os protocolos utilizados pelos componentes reais. Isso torna a validação dos protocolos de comunicação utilizados mais confiável que nos níveis anteriores. Apesar disso, esse modelo ainda não pode ser considerado pronto para síntese, pois utiliza componentes modelados em linguagens de programação de alto-nível e não em linguagens de descrição de componentes de hardware. Por isso, é importante ressaltar que o desenvolvimento de *gateways* para integração de componentes desenvolvidos em diferentes HDLs (*Hardware Description Languages*) são o próximo passo para a obtenção de um modelo simulável mais próximo do circuito final.

A TABELA 6.3 exibe mais informações sobre linguagem e tamanho do código gerado.

Tabela 6.3: Detalhes do experimento 3

Federado	Linguagem do federado	Tamanho do federado (linhas)	Tamanho do *gateway* gerado (linhas)	Template utilizado	Tamanho do XML gerado para config. do EF e EDCB (linhas)
GPS Alerta	Java	345	159	Java	17
Simulador GPS	Java	107	127	Java	12
Display	Java	440	146	Java	8
Driver do display	C++	202	243 *	JNI	17
Teclado	Java	288	146	Java	16
Driver do teclado	C++	364	254 *	JNI	18

* Inclui o código C++ que fornece acesso as funções da *Java Native Interface*.

6.4 Conclusões do Estudo de Caso

A integração dos federados utilizados nos três experimentos foi realizada através da ferramenta de configuração (citada na Seção 5.6) que auxilia na configuração dos embaixadores e na geração dos *gateways*. O tamanho do código gerado para os *gateways* está diretamente relacionado ao número de sinais ou funções da interface do federado, não existindo qualquer relação entre o tamanho do federado e o tamanho do *gateway*. Para os federados descritos em Java foi gerado o *gateway* baseado no *template* Java e para os federados descritos em C++ foi utilizado como base o *template* JNI (*Java Native Interface*), composto por classes Java que comunicam-se com código nativo escrito em C ou C++.

Em todos os experimentos executados, os componentes teclado e *display* foram instalados em uma mesma máquina com o objetivo de permitir, de forma simultânea, a visualização dos resultados da execução do experimento (tarefa do *display*) e o controle de navegação nas funções do GPSAlerta (tarefa do teclado). Apesar destes dois componentes terem sido executados sob uma única JVM, as primitivas para comunicação local (disponibilizadas pela arquitetura DCB) em nenhum momento foram utilizadas, pois o teclado e o *display* não realizam troca direta de mensagens, fazendo com que toda a troca de mensagens entre federados no decorrer dos três experimentos executados fosse feita através da rede.

Os componentes do sistema GPSAlerta foram distribuídos de forma aleatória em três máquinas Pentium III 1Ghz com 128Mb de RAM, conectados a uma rede *Ethernet* de 100Mbits. O comportamento funcional dos modelos distribuídos foi exatamente o mesmo do modelo original não-distribuído, confirmando a correção dos métodos de distribuição e interconexão "virtual" utilizados pela arquitetura DCB.

Nos experimentos realizados não foram levados em conta aspectos temporais dos modelos executados. Isso se deve à utilização de uma camada DCB provisória, desenvolvida especificamente para permitir a execução dos experimentos e a conclusão desse trabalho dentro dos prazos previstos. A camada DCB, responsável pela sincronização, é o principal foco da tese de doutorado citada anteriormente e no decorrer desse trabalho ainda não possuía um protótipo desenvolvido. Assim, a falta de mecanismos de sincronização adequados não permitiu a sincronização entre os modelos, mas sim uma ordenação temporal dos eventos gerados por eles, o que foi suficiente para validar o comportamento dos componentes, assim como as interações entre eles.

Apesar desse estudo de caso não contemplar questões relacionadas ao desempenho em co-simulações distribuídas, pôde-se constatar a visível perda de performance na execução de modelos distribuídos. Isso pode ser atribuído à inevitável sobrecarga que é resultado da integração virtual e distribuição de componentes heterogêneos. Sugestões para otimização de desempenho em co-simulações distribuídas são abordadas na Seção 7.1.

7 TRABALHOS FUTUROS

No decorrer do desenvolvimento deste trabalho de pesquisa foram identificados diferentes caminhos para otimizar a metodologia de co-simulação distribuída empregada na arquitetura DCB e seus componentes. No entanto, devido ao amplo campo de pesquisa que a co-simulação engloba, foi priorizado o estudo e a experimentação de apenas algumas das inúmeras possibilidades existentes. A pesquisa mais aprofundada de alguns dos temas deixados em aberto nesse trabalho e a continuidade do desenvolvimento das idéias já exploradas permitirá, de forma mais ampla, que a arquitetura DCB seja utilizada como uma real alternativa para co-simulação distribuída.

Este trabalho prioriza questões que envolvem interoperabilidade entre modelos descritos em diferentes linguagens e/ou níveis de abstração, deixando outras questões em segundo plano. Por isso, um dos caminhos que podem ser seguidos é o aprimoramento das camadas da arquitetura DCB, levando em conta questões como métodos de sincronização e técnicas para melhorar o desempenho em co-simulações distribuídas. Essas possíveis extensões são mais bem detalhadas nas seções seguintes.

Outra possibilidade a ser explorada é o desenvolvimento de diferentes *templates* de *gateway*, voltados para integração de linguagens específicas para descrição de hardware como, por exemplo, VHDL e SystemC. Essa extensão para a arquitetura DCB é extremamente desejável, pois esse trabalho apesar de ser totalmente voltado para a simulação concorrente de hardware e software, ainda não chega à implementação de um modelo composto por componentes descritos nessas linguagens.

Apesar disso, a exploração desse caminho já foi iniciada, pois a infraestrutura DCB possui módulos que oferecem suporte para comunicação com código nativo via JNI (*Java Native Interface*), que é o primeiro passo para tornar possível a comunicação entre a infraestrutura de co-simulação e os diferentes ambientes voltados para modelagem e simulação de componentes de hardware como Modelsim [MOD2003], VSS Synopsys [SYN2003a] e SystemC (veja detalhes nas Seções 7.2 e 7.3).

7.1 Otimização de desempenho

Da forma como estão implementados os protótipos é possível a cooperação entre federados locais ou distribuídos na rede. Em alguns casos, o grande volume de troca de mensagens com federados remotos pode comprometer o desempenho do modelo de co-simulação. Por isso é necessário abordar alternativas para solucionar esse problema.

Uma possível solução para esse caso poderia ser iniciada com um processo de análise das interações entre os componentes distribuídos e posterior identificação da

ocorrência de grandes volumes de trocas de dados entre um determinado grupo de componentes. Nesse momento, o projetista teria a opção de alocar esses componentes em uma mesma máquina, o que poderia reduzir consideravelmente o tempo consumido pela comunicação via rede. No entanto, essa solução pode tornar-se inviável quando o componente é de propriedade de outro fornecedor ou somente possui uma interface pública para fins de validação.

Outra alternativa para otimização de desempenho na arquitetura DCB seria a utilização de mecanismos semelhantes ao *Selective Focus* [HIN97], tornando possível a troca dinâmica de nível de detalhamento em algumas partes do sistema. Essa abordagem possibilitaria que a comunicação entre determinados componentes do sistema fosse simulada em níveis de abstração mais altos, reduzindo a quantidade de mensagens trocadas entre estes e conseqüentemente incrementando a performance da simulação.

A aplicação das referidas soluções na arquitetura DCB exigiria um estudo mais detalhado, voltado para verificação da viabilidade desses mecanismos, assim como os melhores caminhos para implementação no protótipo.

7.2 Comunicação de simuladores VHDL com *gateways* da arquitetura DCB

O desenvolvimento da VHSIC (*Very High Speed Integrated Circuit*) *Hardware Description Language* (VHDL) foi iniciado em 1983 como uma iniciativa do *US Department of Defense* para criar uma linguagem de especificação para projeto de hardware digital [WIL98]. A semântica do VHDL dá suporte à especificação, simulação e síntese nos níveis RTL (*register-transfer-level*) e comportamental (*behavioral*). A maioria dos sistemas que envolvem o desenvolvimento concorrente de partes de hardware e software utiliza VHDL para descrever o hardware e C/C++ para descrever o software.

Em VHDL, o termo *behavioral description* refere-se a qualquer descrição que expressa operações que irão ocorrer e à troca de informações entre essas operações, sem levar em conta aspectos como alocação de recursos e tempo. Assim, são especificadas de forma explicita operações e seus relacionamentos. Já uma descrição RTL (*register-transfer-level*), além de conter informações sobre todas as operações e seus relacionamentos, também descreve completamente todas as pré-condições para que elas ocorram, geralmente em termos de operações lógicas sobre sinais, os quais são parte do sistema ou são entradas externas de dados.

Nesse trabalho foram abordados diferentes mecanismos para implementar a camada de comunicação (*gateway*) entre a infraestrutura DCB e o federado. No entanto, é necessário mais do que isso para permitir a comunicação entre um modelo executando em um determinado simulador proprietário e a infraestrutura de co-simulação DCB. Por isso, esse capítulo apresenta técnicas gerais para fazer a integração de simuladores VHDL através de código nativo.

O primeiro passo para que um simulador VHDL possa cooperar dentro de uma federação é obtido através da utilização de interfaces compatíveis entre as duas partes envolvidas. A maioria dos simuladores VHDL oferece formas para integrar rotinas em C/C++ com uma descrição VHDL. Essas rotinas foram desenvolvidas com o objetivo inicial de executar operações difíceis de serem obtidas em VHDL e também para

descrever e simular um módulo de software [VAL98]. O acesso a essas rotinas é possível devido à existência do atributo VHDL chamado "*foreign*", definido na especificação da arquitetura VHDL (ANSI/IEEE std. 1076-1993) [SUN97]. O atributo permite que sejam declaradas funções escritas em linguagens diferentes de VHDL, que podem ser executadas durante a simulação [VAL98].

As interfaces que os simuladores oferecem para acesso às linguagens são proprietárias e, na maioria das vezes, são chamadas de FLI (*Foreign Language Interface*) ou CLI (C *Language Interface*). Existem dois principais modelos de comunicação que permitem integração de VHDL com rotinas externas, o modelo mestre-escravo e o modelo distribuído (veja FIGURA 7.1).

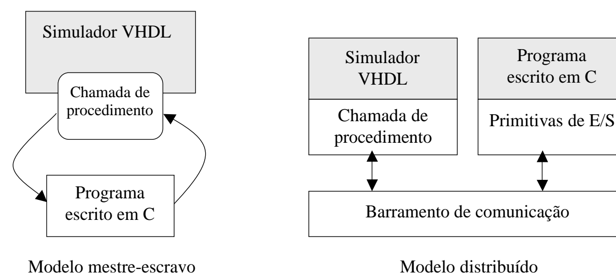


Figura 7.1: Modelos de co-simulação com VHDL.

Diversos autores apresentam suas soluções para permitir a co-simulação com VHDL, entre eles [VAL96], [LIE97], [SUN97], [VAL98], [OYA99]. As duas principais abordagens que poderiam ser utilizadas em futuros protótipos da arquitetura DCB são descritas a seguir.

7.2.1 Modelo mestre-escravo

No modelo de comunicação mestre-escravo, no qual o simulador VHDL é o mestre, o VHDL faz chamadas a rotinas em C e aguarda pela resposta de cada chamada [LIE97]. Nesse modelo, o módulo escravo não pode executar de forma concorrente com o módulo mestre, o que pode ser considerado uma restrição. O modelo mestre-escravo é útil para que procedimentos escritos em C sejam utilizados para especificar o comportamento de uma entidade VHDL como, por exemplo, um processador.

Nas abordagens utilizadas pelo VCI [VAL98] e pelo SIMOO-VSS Synopsys [OYA99] o simulador VHDL é o mestre da simulação. Assim, a aplicação C (escravo) sempre deve esperar por um evento VHDL para enviar atualizações de valores. Essa sincronização entre a aplicação C e o simulador VHDL foi implementada através de um protocolo de comunicação *half-duplex*, baseado na troca contínua de atualizações [VAL96]. Assim, um *token* é associado com cada atualização, permitindo que o programa C (receptor) envie uma resposta no próximo passo de execução. Após enviar uma atualização, não é permitido o envio de outra mensagem antes que seja recebido outro *token*. Esse mecanismo assegura que um e somente um evento possa ser entregue ao programa C ou ao VHDL em um instante de tempo, assegurando a sincronização dos eventos [LIE97].

No módulo de comunicação gerado pela ferramenta VCI, quando os programas escritos em C interagem com o conjunto de módulos de hardware, primitivas de E/S são executadas para permitir a execução de operações sobre as portas de hardware. Já no lado do VHDL, o programa C é conectado ao barramento de hardware através de uma entidade VHDL que encapsula as portas utilizadas pelas primitivas de E/S.

7.2.2 Modelo distribuído

No modelo de comunicação distribuído não existe a noção de mestre e escravo e isso ameniza algumas limitações oferecidas pelo modelo anterior. Normalmente, o modelo distribuído é implementado sobre algum tipo de mecanismo de comunicação interprocessos como *sockets* ou mensagens IPC (*Inter-Process Communication*). Neste modelo, as rotinas externas são totalmente responsáveis pelas tarefas de conversão de dados, sincronização e propagação de eventos entre o simulador VHDL e o canal de comunicação [VAL96].

Para executar uma co-simulação VHDL baseada em eventos utilizando diferentes simuladores distribuídos, é necessário definir um protocolo de sincronização entre esses simuladores. Normalmente, pode ser utilizada uma abordagem conservativa, de modo que o relógio do simulador VHDL não pode estar na frente do relógio global [SUN97]. No entanto, a implementação de protocolos otimistas também é possível, mas é necessária a existência de funções adicionais que permitam salvar todos os valores de sinais (inclusive os já escalonados) e também restaurá-los. Essas funções impõem custos ao tempo de execução e ao consumo de memória [YOO97].

Nesse tipo de modelo também deve ser levado em conta o risco de bloqueio no caso da existência de mais de um *link* de comunicação entre hardware e software. Contudo, esse problema pode ser evitado através do gerenciamento cuidadoso da ordem de disparo dos módulos receptores. A solução seria a inclusão de uma entidade VHDL, chamada de "nodo mestre" [VAL96] [SUN97], que serializa os disparos dos nodos de comunicação.

7.3 Comunicação de modelos SystemC com *gateways* da arquitetura DCB

SystemC [SYS2002] é basicamente uma plataforma de modelagem, composta por um conjunto de bibliotecas de classes C++, que oferece um *kernel* de simulação. A plataforma SystemC disponibiliza classes para especificação de blocos de hardware, software e canais de comunicação em diferentes níveis de abstração (ex. *system level*, *behavioral level*, *register transfer level*).

Normalmente, os tradicionais ambientes de co-simulação são baseados em descrições multi-linguagem, ou seja, utilizam linguagens de descrição de hardware (ex. VHDL) para descrever o hardware e C/C++ (ou linguagem similar) para descrever o software. Uma das motivações para o desenvolvimento do SystemC foi o desejo de utilizar uma única linguagem para especificar simultaneamente componentes de hardware e software, assim como suas interações.

O ambiente SystemC é baseado no conceito de módulos, que podem conter um ou mais processos ou ainda outros módulos, permitindo uma organização hierárquica dos

componentes. Os processos descrevem o comportamento de um determinado módulo e oferecem métodos para expressar concorrência [CON2002]. Já a comunicação entre os módulos é feita através de portas, que são conectadas a determinados canais. Os canais implementam uma ou mais interfaces, que são simplesmente um conjunto de métodos (funções). Desse modo, um processo pode acessar a interface de um canal através de uma porta do seu módulo pai.

Uma das características do SystemC é o suporte a múltiplos níveis de abstração. Essa característica permite que a fase de modelagem seja iniciada com um modelo descrito no mais alto nível de abstração (nível funcional), que sofrerá um processo de refinamento sucessivo, através da agregação de detalhes de implementação, até chegar ao mais baixo nível de abstração possível, o nível RT (*Register Transfer*). Nos modelos SystemC, o processo de refinamento pode ser aplicado ao conteúdo dos componentes, à comunicação, aos tipos de dados e aos protocolos [ARN2000].

A integração de modelos descritos em SystemC com a infraestrutura de co-simulação DCB pode ser feita de duas principais formas: através de um mecanismo de comunicação direta entre Java e C/C++ (ex. interface JNI) ou através da utilização de um modelo de comunicação distribuído (ex. *sockets* TCP). As principais implicações para essas duas alternativas são analisadas a seguir.

A integração através de comunicação direta entre linguagens é o caminho mais natural, pois a implementação dessa alternativa poderia partir da extensão/reuso de um modelo de integração já existente (previamente desenvolvido), que é o caso do *template* JNI (veja seção 5.5.3). Essa abordagem exigirá modificações no *template* JNI e também no módulo da ferramenta de configuração que é responsável pela geração do *gateway*. Essas modificações são necessárias para adequação do *gateway* às características do modelo de interface utilizado pelo SystemC. Além disso, também será necessária a criação de módulos de interface para o modelo SystemC, que seriam responsáveis pela execução de tarefas relacionadas à tradução de dados e sincronização.

A outra alternativa de integração seria a utilização de um modelo distribuído semelhante ao proposto na seção anterior para integrar modelos VHDL. Essa abordagem exigiria a utilização de algum tipo de mecanismo interprocessos como *sockets*. Em [BEN2003] é apresentado um experimento de co-simulação que utiliza componentes SystemC e VHDL executando de forma distribuída. Esse modelo utiliza um *wrapper* para o barramento de comunicação descrito em SystemC. A comunicação entre o *wrapper* e outros componentes do sistema é feita através de um mecanismo interprocessos que é responsável por questões de sincronização e tradução de informações. A solução descrita ainda utiliza uma interface, chamada de *virtual socket interface*, que realiza a tradução de eventos gerados pelo *Instruction-Set Simulator* (ISS) em transações válidas para o barramento utilizado pelos módulos SystemC.

A utilização de qualquer uma das abordagens descritas acima exige uma análise prévia da necessidade de realizar conversões semânticas entre os modelos. Esse tipo de conversão pode ser necessária quando são utilizados modelos que possuem dinâmica e/ou modo de execução heterogêneos, o que pode influir na execução da co-simulação como um todo. Um exemplo que poderia caracterizar essa situação seria um modelo de co-simulação onde um modelo SystemC baseado em ciclos (*cycle-based*) interage com um modelo VHDL baseado em eventos (*event-driven*), o que normalmente exigirá a implementação de um mecanismo de conversão apropriado.

Analisando os recursos de comunicação oferecidos por VHDL e SystemC, assim como as técnicas que poderiam ser empregadas para integração dessas ferramentas com

gateways da arquitetura DCB, pode se chegar a conclusão de que o esforço para integração de modelos descritos em SystemC seria relativamente menor em comparação com VHDL. Isso pode ser atribuído aos seguintes fatores:

- SystemC é um ambiente aberto, ao contrário de VHDL, onde normalmente são utilizadas ferramentas proprietárias, que possuem interfaces de acesso distintas entre si (ex. CLI, FLI, etc);
- Por ser basicamente uma descrição em C/C++, SystemC permite a definição de um método de acesso direto e padronizado (via interface JNI) aos objetos.

O desenvolvimento de um *template* SystemC, partindo de uma versão modificada do *template* JNI, e de um novo módulo da ferramenta de configuração para geração automática / semi-automática das interfaces é a solução ideal. A principal vantagem dessa abordagem é não utilizar mecanismos interprocessos, o que evita o aumento da sobrecarga gerada pela comunicação que já é considerada alta em simulações distribuídas.

8 CONCLUSÃO

Esse trabalho apresenta soluções para algumas das carências existentes na área de co-simulação. As maiores contribuições estão relacionadas à especificação e construção de mecanismos genéricos e extensíveis que oferecem suporte à co-simulação distribuída e heterogênea. Esses mecanismos auxiliam nas tarefas que envolvem a interconexão virtual e execução concorrente de componentes heterogêneos, permitindo a cooperação transparente entre componentes locais ou remotos.

O trabalho especifica uma ferramenta de apoio que auxilia na construção das interfaces de co-simulação e na configuração da infraestrutura de co-simulação. O mecanismo responsável pela interoperabilidade entre linguagens (*gateway*) é gerado a partir de *templates* configuráveis com auxílio da ferramenta de configuração. A utilização dessas ferramentas em um ambiente distribuído flexibiliza o processo de projeto de sistemas, pois torna mais simples a substituição de componentes, diminuindo o esforço empregado no processo sucessivo de refinamento e experimentação. Dessa forma, o projetista pode executar experimentos construídos a partir de diferentes configurações de componentes, até chegar à configuração mais adequada para o seu modelo. Isso pode reduzir de forma considerável o tempo de validação e o custo de desenvolvimento de um determinado projeto.

Os mecanismos especificados encontram-se organizados em uma arquitetura modular que permite a integração de componentes sem que estes precisem se preocupar com aspectos relacionados à distribuição ou sincronização. A infraestrutura DCB está preparada para futuras extensões, como a inclusão de mecanismos de suporte à sincronização (o modelo que suporta a execução em modo síncrono já está implementado em outro trabalho) e ao gerenciamento de dados (gerenciamento de propriedade de atributos). Além disso, a ferramenta de configuração prevê a adição de novos *templates* e módulos específicos para configuração dos mesmos, construídos com o objetivo de permitir a comunicação com novas linguagens e/ou simuladores.

A disponibilidade de ambientes que disponibilizem recursos para integração de componentes heterogêneos (descritos em diferentes linguagens e/ou plataformas) oferece ao projetista uma maior flexibilidade durante o projeto do sistema, pois facilita o reuso de componentes já existentes e ainda dá mais liberdade na escolha da linguagem ou simulador mais apropriado para cada componente. O reuso, além de evitar o esforço de recodificação de determinados componentes, permite que o projetista concentre-se principalmente nas tarefas de integração e validação do sistema como um todo. Esses são fatores cruciais para a diminuição do tempo total de desenvolvimento do sistema, uma exigência atual do mercado, que pressiona pelo lançamento de novos produtos em um tempo cada vez menor. A liberdade de escolha do projetista também precisa ser levada em consideração, pois não é desejável que este fique preso a uma única ferramenta ou linguagem para descrição dos componentes, pois muitas vezes é

necessário utilizar uma linguagem específica para descrever um determinado componente. Esse fator está relacionado com a capacidade de representação da determinada linguagem ou até mesmo à facilidade de construir o componente utilizando os recursos oferecidos pela mesma.

A abordagem para execução de co-simulações de forma distribuída oferecida pela arquitetura DCB permite que componentes desenvolvidos por diferentes equipes e muitas vezes localizados em máquinas remotas sejam avaliados por meio de simulação (através da adição de interfaces adequadas). Além disso, a propriedade intelectual do componente é preservada, devido ao modelo de simulação distribuída baseado na troca de mensagens, o que certamente é um fator importante para desenvolvedores que desejam disponibilizar seus componentes para experimentação e posterior comercialização. Esses aspectos, somados à existência de bancos de dados para armazenamento e busca de componentes, poderiam aumentar consideravelmente a gama de opções para o projetista no momento da escolha dos componentes para o seu sistema.

Atualmente a área de co-simulação distribuída está carente de mecanismos que atendam as reais necessidades dos projetistas. A pressão exercida pelo mercado faz com que projetistas busquem por novas metodologias, normalmente baseadas no reuso de componentes, com o objetivo de reduzir o tempo de construção dos sistemas. No entanto, obter componentes que sejam compatíveis com as necessidades do sistema pode ser uma tarefa difícil, pois muitas vezes não existe a possibilidade ou meios para avaliar o componente desejado de forma adequada. Por isso a arquitetura DCB é apresentada como uma alternativa para resolver alguns dos problemas atualmente encontrados na área de co-simulação. A solução descrita nesse trabalho disponibiliza soluções de suporte à interconexão de componentes heterogêneos e posterior validação através de simulação, além de levar em consideração fatores como reuso e preservação da propriedade intelectual.

REFERÊNCIAS

[ACC2003] ACCELLERA. **Verilog**. Disponível em: <http://www.accellera.org>. Acesso em: 07 maio 2003.

[ADA96] ADAMS, J. K.; THOMAS, D. E. The Design of Mixed Hardware/Software Systems. In: DESIGN AUTOMATION CONFERENCE, 1996. **Proceedings…** New York: ACM, 1996. p. 515-520.

[ARN2000] ARNOUT, G. SystemC Standard. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2000. **Proceedings…** New York: ACM, 2000. p.573-577.

[BAL97] BALARIN, F. et al. **Hardware-Software Co-Design of Embedded Systems:** the polis approach. Boston: Kluwer Academic, 1997.

[BEN2003] BENINI, L. et al. SystemC Cosimulation and Emulation of Multiprocessor SoC Designs. **Computer**, Los Alamitos, v. 36, n. 4, p.53-59, Apr. 2003.

[BIS97] BISHOP, W. D.; LOUCKS, W. M. A Heterogeneous Environment for Hardware/Software Cosimulation. In: SIMULATION SYMPOSIUM, 1997. **Proceedings…** Los Alamitos: IEEE Computer Society, 1997. p. 14-22.

[BRA99] BRASSÉ, M.; KUIJPERS, N. Standardising Distributed Simulations: The High Level Architecture. **Xootic Magazine**, [S. l.], v. 7, n. 1, p. 16-24, July 1999.

[BRY77] BRYANT, R. E. **Simulation of Packet Communication Architecture Computer Systems**. [S. l.]: MIT Laboratory for Computer Science, 1977. (Technical Report TR-188)

[BUC94] BUCHENRIEDER, K.; ROZENBLIT J. W. Codesign: an overview. In: BUCHENRIEDER, K.; ROZENBLIT J. W. **Codesign**: computer-aided software/hardware engineering. New York: IEEE, 1994. p. 1-15.

[CES2002] CESÁRIO, W. O. et al. Component-based Design Approach for Multicore SoCs. In: DESIGN AUTOMATION CONFERENCE, 2002. **Proceedings…** New York: ACM, 2002. p. 789-794.

[CHA79] CHANDY, K. M.; MISRA J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. **IEEE Transactions on Software Engineering**, Los Alamitos, v. 5, n. 5, p. 440-452, Sept. 1979.

[CHI94] CHIODO, M. et al. Hardware-Software Codesign of Embedded Systems. **IEEE Micro**, Los Alamitos, v. 14, n. 4, p. 26-36, Aug. 1994.

[CHO99] CHOU, P. et al. IPChinook: An Integrated IP-Based Design Framework for Distributed Embedded Systems. In: DESIGN AUTOMATION CONFERENCE, 1999. **Proceedings…** New York: ACM, 1999. p. 44-49.

[CON2002] CONNELL, J.; JOHNSON, B. **Early Hardware/Software Integration Using SystemC 2.0**. Disponível em: <http://www.synopsys.com/products/cocentric_studio/esc_paper_552.pdf>. Acesso em: 15 dez. 2002.

[COP97] COPSTEIN, B.; WAGNER, F. R.; PEREIRA, C. E. SIMOO – An Environment for the Object-Oriented Discrete Simulation. In: EUROPEAN SIMULATION SYMPOSIUM, ESS, 9., Passau, Germany. **Simulation in Industry**. Ghent: SCS, 1997. p. 21-25.

[COU95] COUMERI, S. L.; THOMAS D. E. A Simulation Environment for Hardware-Software Codesign. In: DESIGN AUTOMATION CONFERENCE, 1995. **Proceedings…** New York: ACM, 1995. p. 58-63.

[DAH97] DAHMANN, J. S.; FUJIMOTO, R. M.; WEATHERLY, R. M. The Department of Defense High Level Architecture. In: WINTER SIMULATION CONFERENCE, 1997. **Proceedings…** New York: ACM, 1997. p. 142-149.

[DAH98] DAHMANN, J. S.; FUJIMOTO, R. M.; WEATHERLY, R. M. The DoD High Level Architecture for Simulation: An Update. In: WINTER SIMULATION CONFERENCE, 1998. **Proceedings…** New York: ACM, 1998. p. 797-804.

[DZI2003] DZIRI, M. A. et al. Combining Architecture Exploration and a Path to Implementation to Build a Complete SoC Design Flow from System Specification to RTL. In: ASIA PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2003. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003. p. 21-24.

[EDW97] EDWARDS, S. et al. Design of Embedded Systems: Formal Models, Validation, and Synthesis. **Proceedings of the IEEE**, New York, v. 85, n. 3, p. 366-390, Mar. 1997.

[ERN98] ERNST, R. Codesign of Embedded Systems: Status and Trends. **Design & Test of Computers**, Los Alamitos, v. 15, n. 2, p. 44-54, June 1998.

[FER95] FERSCHA, A. Parallel and Distributed Simulation of Discrete Event Systems. In: ZOMAYA, A. Y. H. **Parallel and Distributed Computing Handbook**. New York: McGraw-Hill, 1995. p. 1003-1041.

[FOR98] FORNACIARI, W. et al. Power Estimation of Embedded Systems: A Hardware/Software Codesign Approach. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Los Alamitos, v. 6, n. 2, p. 266-275, June 1998.

[FUJ90] FUJIMOTO, R. M. Parallel Discrete Event Simulation. **Communications of the ACM**, New York, v. 33, n. 10, p. 30-53, Oct. 1990.

[FUJ96] FUJIMOTO, R. M.; WEATHERLY, R. M. HLA Time Management and DIS. In: WORKSHOP ON STANDARDS FOR THE INTEROPERABILITY OF DISTRIBUTED SIMULATION, 1996. **Proceedings…** [S. l.: s. n.], 1996. p. 615-628.

[FUJ99] FUJIMOTO, R. M. Parallel and Distributed Simulation. In: WINTER SIMULATION CONFERENCE, 1999. **Proceedings…** New York: ACM, 1999. p. 122-131.

[FUJ2001] FUJIMOTO, R. M. Parallel and Distributed Simulation Systems. In: WINTER SIMULATION CONFERENCE, 2001. **Proceedings…** New York: ACM, 2001. p. 147-157.

[HAN98] HANSEN, C.; KUNZMANN, A.; ROSENTIEL, W. Verification by Simulation Comparison using Interface Synthesis. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 1998. **Proceedings…** Los Alamitos: IEEE Computer Society, 1998. p. 436-445.

[HEL97] HELAIHEL. R.; OLUKOTUN, K. Java as a Specification Language for Hardware-Software Systems. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, ICCAD, 1997. **Proceedings…** Los Alamitos: IEEE Computer Society, 1997. p. 690-697.

[HES99] HESSEL, F. et al. MCI - Multilanguage Distributed Co-Simulation Tool. In: RAMMIG, F. **Distributed and Parallel Embedded Systems**. Boston: Kluwer Academic, 1999. p. 191-200.

[HIN96] HINES, K. **Pia:** A Framework for Embedded System Co-Simulation with Dynamic Communication Support. [S. l.]: University of Washington, 1996. (Technical Report UW-CSE-96-11-04).

[HIN97] HINES, K.; BORRIELO, G. Selective Focus as a Means of Improving Geographically Distributed Embedded System Co-Simulation. In: INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 8., 1997. **Proceedings…** Los Alamitos: IEEE Computer Society, 1997. p. 58-62.

[HIN97a] HINES, K.; BORRIELO, G. Dynamic Communication Models in Embedded System Co-Simulation. In: DESIGN AUTOMATION CONFERENCE, 1997. **Proceedings…** New York: ACM, 1997. p. 395-400.

[HIN98] HINES, K.; BORRIELLO, G. A Geographically Distributed Framework for Embedded System Design and Validation. In: DESIGN AUTOMATION CONFERENCE, 1998. **Proceedings…** New York: ACM, 1998. p. 140-145.

[HUB98] HÜBERT, H. **A Survey of HW/SW Cosimulation Techniques and Tools**. 1998. Master Thesis. Royal Institute of Technology, Stockholm.

[IEE2000] IEEE. **Std 1076-2000:** VHDL Language Reference Manual. New York, 2000.

[ITO99] ITO, S. A.; CARRO, L.; JACOBI, R. P. Designing a Java Microcontroller to Specific Applications. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 1999. **Proceedings…** Los Alamitos: IEEE Computer Society, 1999. p. 12-15.

[ITO2000] ITO, S. A.; CARRO, L.; JACOBI, R. P. System Design Based on Single Language and Single-Chip Java ASIP Microcontroller. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2000. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000. p. 703-709.

[ITO2001] ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **Design & Test of Computers**, Los Alamitos, v. 18, n. 5, p. 100-110, Oct. 2001.

[JEF85] JEFFERSON, D. R. Virtual Time. **ACM Transactions on Programming Languages and Systems**, New York, v. 7, n. 3, p. 404-425, July 1985.

[JEN97] JENSE, G. J.; KUIJPERS, N. H. L.; DUMAY, A. C. M. DIS and HLA: Connecting People, Simulations and Simulators in the Military, Space and Civil Domains. In: INTERNATIONAL ASTRONAUTICAL CONGRESS, 1997. **Proceedings…** [S. l.]: AIAA, 1997.

[JER99] JERRAYA, A. A. et al. Multilanguage Specification for System Design and Codesign. In: JERRAYA, A. A.; MERMET, J. **System Level Synthesis**. Boston: Kluwer Academic, 1999.

[KIM95] KIM, Y. et al. An Integrated Hardware-Software Cosimulation Environment for Heterogeneous Systems Prototyping. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 1995. **Proceedings…** Los Alamitos: IEEE Computer Society, 1995. p. 101-106.

[KUH99] KUHN, T.; ROSENSTIEL, W.; KEBSCHULL, U. Description and Simulation of Hardware/Software Systems with Java. In: DESIGN AUTOMATION CONFERENCE, 1999. **Proceedings…** New York: ACM, 1999. p.790-793.

[LEE2001] LEE, E. A. **Overview of the Ptolemy Project**. [S. l.]: Department of Electrical Engineering and Computer Science, University of California, 2001. (Technical Memorandum UCB/ERL M01/11).

[LIE97] LIEM, C. et al. System-on-a-chip Cosimulation and Compilation. **Design and Test of Computers**, Los Alamitos: IEEE Computer Society, v. 14, n. 2, p. 16-25, June 1997.

[LIS2002] LISBOA, C. A. L. **Sistema GPSAlerta – Relatório da Implementação**. 2002. Trabalho da disciplina Tópicos Especiais em Computação VII (Diferentes Aspectos de Sistemas em Silício). Instituto de informática – UFRGS, Porto Alegre.

[MEL2001] MELLO, B. A. **Construção de um Mecanismo de Suporte à Co-Simulação em Ambientes Distribuídos**. 2001. Proposta de Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[MEL2002] B.A.MELLO, F.R.WAGNER A Standardized Co-Simulation Backbone. In: CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEMS-ON-CHIP, VLSI-SOC, 11., 2001. **SOC Design Methodologies**. Boston: Kluwer Academic, 2001. p.181 – 192.

[MEN2003] MENTOR GRAPHICS. **Seamless CVE**. Disponível em: <http://www.mentor.com/seamless>. Acesso em: 14 fev. 2003.

[MOD2003] MODEL TECHNOLOGY. **ModelSim PE/SE**. Disponível em: <http://www.model.com/products/modelsim_pe_se.asp>. Acesso em: 26 jul. 2003.

[NAN99] NANCE, R. E. Distributed Simulation with Federated Models: Expectations, Realizations and Limitations. In: WINTER SIMULATION CONFERENCE, 1999. **Proceedings…** New York: ACM, 1999. p. 1026-1031.

[NIC94] NICOL, D.; FUJIMOTO, R. Parallel Simulation Today. **Annals of Operations Research**, [S. l.], v. 53, p. 249-286, Dec. 1994.

[NIC96] NICOL, D. M. Principles of Conservative Parallel Simulation. In: WINTER SIMULATION CONFERENCE, 1996. **Proceedings…** New York: ACM, 1996. p. 128-135.

[ORT99] ORTEGA, R. B.; LAVAGNO, L.; BORRIELLO, B. Models and Methods for HW/SW Intellectual Property Interfacing. In: JERRAYA, A. A.; MERMET, J. **System Level Synthesis**. Boston: Kluwer Academic, 1999.

[OYA99] OYAMADA, M. S.; WAGNER, F. R. Ambiente para Co-Simulação SIMOO - VSS Synopsys. In: WORKSHOP IBERCHIP, 5., 1999, Lima, Pe. **Memorias**. [Lima: Hozlo S. R. L.], 1999. p. 182-187.

[PAG99] PAGE, E. H. Beyond Speedup: PADS, the HLA and Web-Based Simulation. In: WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, 1999. **Proceedings…** New York: ACM, 1999. p. 2-9.

[PAN2000] PANIGRAHI, D.; TAYLOR, C. N.; DEY S. Interface Based Hardware/Software Validation of a System-on-chip. In: HIGH-LEVEL VALIDATION AND TEST WORKSHOP, HLDVT, 2000. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000. p. 53-58.

[ROD2003] RODRIGUES, R.; LISKOV, B.; SHRIRA, L. **The Design of a Robust Peer-to-Peer System**. Disponível em: <www.pmg.lcs.mit.edu/~rodrigo/ ew02-robust.pdf>. Acesso em: 30 ago. 2003.

[ROW94] ROWSON, J. Hardware/Software Co-Simulation. In: DESIGN AUTOMATION CONFERENCE, 1994. **Proceedings…** New York: ACM, 1994. p. 439-440.

[ROW97] ROWSON, J. A.; SANGIOVANNI-VINCENTELLI, A. Interface-Based Design. In: DESIGN AUTOMATION CONFERENCE, 1997. **Proceedings…** New York: ACM, 1997. p. 178-183.

[SAS2002] SASHIMI PROJECT HOME PAGE. **Sashimi**. Disponível em: <http://www.inf.ufrgs.br/sashimi/software.html>. Acesso em: 15 out. 2002.

[SCH95] SCHMERLER, S.; TANURHAN, Y.; MÜLLER-GLASER, K. D. A Backplane Approach for Cosimulation in High-level System Specification Environments. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, EURO-DAC, 1995. **Proceedings…** Los Alamitos: IEEE Computer Society, 1995. p. 262-267.

[STR98] STRAßBURGER, S.; SCHULZE, T.; KLEIN, U.; HENRIKSEN, J. O. Internet-Based Simulation using Off-the-shelf Simulation Tools and HLA. In: WINTER SIMULATION CONFERENCE, 1998. **Proceedings…** Los Alamitos: IEEE Computer Society, 1998. p. 1669-1676.

[STR2000] STRAßBURGER, S. **Distributed Simulation based on High Level Architecture in Civilian Application Domains**. 2000. Ph.D. Thesis. Magdeburg, Germany.

[SUN97] SUNG, W.; OH M.; HA S. Interface Design of VHDL Simulation for Hardware-Software Cosimulation. In: ASIA PACIFIC CONFERENCE ON HARDWARE DESCRIPTION LANGUAGES, APCHDL, 1997. **Proceedings…** [S. l.: s. n.], 1997. p. 43-49.

[SYN2003a] SYNOPSYS. **VHDL System Simulator (VSS)**. Disponível em: <http://www.synopsys.com/products/simulation/vss_cs.html>. Acesso em: 26 jul. 2003.

[SYN2003b] SYNOPSYS. **CoCentric System Studio**. Disponível em: <http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html>. Acesso em: 20 jul. 2003.

[SYS2002] SYSTEMC. **SystemC**. Disponível em: <http://www.systemc.org>. Acesso em: 23 dez. 2002.

[THO98] THONDUGULAN, N. **Unsynchronized Parallel Discrete Event Simulation**. 1998. Master Thesis, Department of Electrical and Computer Engineering and Computer Science of College of Engineering, University of Cincinnati.

[USD98] U. S. DEPARTMENT OF DEFENSE. **High Level Architecture Rules Version 1.3**. 1998. Disponível em: <https://www.dmso.mil/public/transition/hla/techspecs/>. Acesso em: 12 maio 2002.

[VAC99] VACCARO, G. L. R. **Simulação Paralela e Distribuída com vistas ao Co-Design**. 1999. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[VAL96] VALDERRAMA, C. A. et al. Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems: an Industrial Experience. In: WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 1996. **Proceedings…** Los Alamitos: IEEE Computer Society, 1996. p. 72-77.

[VAL98] VALDERRAMA, C. A. et al. Automatic VHDL-C Interface Generation for Distributed Cosimulation: Application to Large Design Examples. **Design Automation for Embedded Systems**, Boston, v. 3, n. 2/3, p. 199-217, Mar. 1998.

[VER96] VERCAUTEREN, S.; LIN, B.; MAN H. D. Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications. In: DESIGN AUTOMATION CONFERENCE, 1996. **Proceedings…** New York: ACM, 1996. p. 521-526.

[WAG99] WAGNER, F. R.; OYAMADA, M.; CARRO, L.; KREUTZ, M. Object-Oriented Modeling and Co-Simulation of Embedded Electronic Systems. In: VERY LARGE SCALE INTEGRATION, VLSI, 1999. **Proceedings…** Boston: Kluwer Academic, 1999. p. 497-508.

[WIL98] WILLIAMSON, M. C. **Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications**. 1998. PhD. Dissertation. University of California, Berkeley.

[WIL99] WILSEY, P.A. Web-Based Analysis and Distributed IP. In: WINTER SIMULATION CONFERENCE, 1999. **Proceedings…** New York: ACM, 1999. p. 1445-1453.

[WOL94] WOLF, W. Hardware-Software Co-Design of Embedded Systems. **Proceedings of the IEEE**, Los Alamitos, v. 82, n. 7, p. 967-989, July 1994.

[YOO97] YOO, S.; CHOI, K. Optimistic Timed HW-SW Cosimulation. In: ASIA PACIFIC CONFERENCE ON HARDWARE DESCRIPTION LANGUAGES, APCHDL, 1997. **Proceedings…** [S. l.: s. n.], 1997. p. 39-42.

ANEXO A TEMPLATE JAVA

(gateway.java)

```
import java.io.*;
/////////////////////////////// INICIO CLASSE GATEWAY ////////////////////////////////// /////

public class Gateway
{
     public ApplicationDCB App;
     public YOUR_FEDERATE_CLASS_HERE Fed;  // Fornece acesso aos metodos do Federado

     //////////////////////////////////////////////////////////////////////////////////////////
     public Gateway(ApplicationDCB pointer)
     {
          App = pointer;

          System.out.println("Gateway Inicializado...");

          Fed = new YOUR_FEDERATE_CLASS_HERE(this);
     }

     public Gateway()
     {
     }

     ///////////////////////////////////////////////////////////////////////////////////////// /
     public void ProtocolConverter(int AttributeID)
     {
          ////////////////////// INÍCIO GERAÇÃO DE CÓDIGO //////////////////////////////////


          /////////////////////// FIM GERAÇÃO DE CÓDIGO ////////////////////////////////////
     }

     /////////////////////////////// INICIO TRADUTOR ////////////////////////////////////////

     public long ToLong(String value)
     {
          return Long.parseLong(value);
     }

     public double ToDouble (String value)
     {
          return Double.parseDouble(value);
     }

     public float ToFloat(String value)
     {
          return Float.parseFloat(value);
     }

     public int ToInt(String value)
     {
          return Integer.parseInt(value);
     }

     public char ToChar(String value)
     {
          return value.charAt(0);
     }

     public char[] ToCharArray(String value)
     {
          return value.toCharArray();
     }

     public boolean ToBoolean (String value)
     {
```

```
          boolean bool = false;
          String val = value.toUpperCase();

          if (val.compareTo("TRUE") == 0)
               bool = true;

          return bool;
     }

     public void UpdateAttribute(String Name, String Value)
     {
          App.NewEDCB.Update(Name,Value);
     }

     public void UpdateAttribute(String Name, int Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value));
     }

     public void UpdateAttribute(String Name, boolean Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value ));
     }

     public void UpdateAttribute(String Name, float Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value));
     }

     public void UpdateAttribute(String Name, double Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value));
     }

     public void UpdateAttribute(String Name, long Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value));
     }

     public void UpdateAttribute(String Name, char Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value));
     }

     public void UpdateAttribute(String Name, char[] Value)
     {
          App.NewEDCB.Update(Name,String.valueOf(Value));
     }
     ////////////////////////////////// FIM TRADUTOR /////////////////////////////////////////
}
////////////////////////////////// FIM CLASSE GATEWAY //////////////////////////////////////
```

ANEXO B TEMPLATE JNI

(interface.cpp)

```
#include <stdio.h>
#include <conio.h>
#include "jni.h"
#include "Gateway.h"

/////// INÍCIO DECLARAÇÃO DO HEADER PARA ACESSO AS FUNÇÕES DO FEDERADO (GERAÇÃO AUTOMÁTICA) /////////

///////// FIM DECLARAÇÃO DO HEADER PARA ACESSO AS FUNÇÕES DO FEDERADO (GERAÇÃO AUTOMÁTICA) //////////

#define PATH_SEPARATOR ';'
#define USER_CLASSPATH "."

// DECLARACAO DE VARIAVEIS JNI
static JNIEnv *env;
static JavaVM *jvm;

//////////////// INÍCIO DECLARACAO DE FUNCOES JNI (GERAÇÃO AUTOMÁTICA) /////////////////////////////


//////////////////////////// FIM DECLARACAO DE FUNCOES JNI /////////////////////////////////////////

///////////// INÍCIO DECLARACAO DE VARIAVEIS DE CONTROLE (GERAÇÃO AUTOMÁTICA) //////////////////////


/////////////////////// FIM DECLARACAO DE VARIAVEIS DE CONTROLE ////////////////////////////////////

////////////////////////////////////////// INICIO MAIN /////////////////////////////////////////////
int main()
{
    jint res;
    jclass cls;
    jclass Gatewaycls;
    jmethodID mid;
    jstring jstr;
    jclass stringClass;
    jobjectArray args;

#ifdef JNI_VERSION_1_2

    JavaVMInitArgs vm_args;
    JavaVMOption options[1];
    options[0].optionString = "-Djava.class.path=" USER_CLASSPATH;
    vm_args.version = 0x00010002;
    vm_args.options = options;
    vm_args.nOptions = 1;
    vm_args.ignoreUnrecognized = JNI_TRUE;
     res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

#else

    JDK1_1InitArgs vm_args;
    char classpath[1024];
    vm_args.version = 0x00010001;
    JNI_GetDefaultJavaVMInitArgs(&vm_args);
    sprintf(classpath, "%s%c%s",vm_args.classpath, PATH_SEPARATOR, USER_CLASSPATH);
    vm_args.classpath = classpath;
     res = JNI_CreateJavaVM(&jvm, &env, &vm_args);

#endif /* JNI_VERSION_1_2 */

     if (res < 0) {
        fprintf(stderr, "Can't create Java VM\n");
        goto destroy;
    }
```

```
    cls = env->FindClass("Start");
    if (cls == 0) {
        goto destroy;
    }

    Gatewaycls = env->FindClass("Gateway");
    if (cls == 0) {
        goto destroy;
    }
   ///////////// INÍCIO REGISTRO DOS MÉTODOS NATIVOS (GERAÇÃO AUTOMÁTICA) //////////////////////////


   /////////////////// FIM REGISTRO DOS MÉTODOS NATIVOS ////////////////////////////////////////////

    mid = env->GetStaticMethodID(cls, "main","([Ljava/lang/String;)V");
    if (mid == 0) {
        goto destroy;
    }
    jstr = env->NewStringUTF(" ");
    if (jstr == 0) {
        goto destroy;
    }
    stringClass = env->FindClass("java/lang/String");
    args = env->NewObjectArray(1, stringClass, jstr);
    if (args == 0) {
        goto destroy;
    }
    env->CallStaticVoidMethod(cls, mid, args);

destroy:
    if (env->ExceptionOccurred()) {
        env->ExceptionDescribe();
    }

    jvm->DestroyJavaVM();

    return 0;
}
////////////////////////////////////////// FIM MAIN //////////////////////////////////////////////////

/////////////// INICIO WRAPPER PARA O CODIGO DO FEDERADO (GERAÇÃO AUTOMÁTICA) //////////////////////

//////////////////////// FIM WRAPPER PARA O CODIGO DO FEDERADO ////////////////////////////////////

////////////// INICIO FUNCOES DE CALLBACK PARA A INTERFACE DO FEDERADO (GERAÇÃO AUTOMÁTICA) /////////

//////////////////////// FIM FUNCOES DE CALLBACK PARA A INTERFACE DO  FEDERADO //////////////////////
```

(gateway.java)

```
import java.io.*;
/////////////////////////////// INICIO CLASSE GATEWAY ///////////////////////////////////////////////

public class Gateway
{
     public static ApplicationDCB App;

     //////////////////////////////////////////////////////////////////////////////////////
     public Gateway(ApplicationDCB pointer)
     {
          App = pointer;

          System.out.println("Gateway Inicializado...");
     }

     public Gateway()
     {
     }

     //////////////////////////////////////////////////////////////////////////////////////
     public synchronized void ProtocolConverter(int AttributeID)
     {
          ////////////////////// INÍCIO GERAÇÃO DE CÓDIGO ///////////////////////////////////


          ///////////////////////// FIM GERAÇÃO DE CÓDIGO ////////////////////////////////////
     }


     //////////////// INICIO FUNÇÕES NATIVAS DO FEDERADO (GERAÇÃO AUTOMÁTICA) ///////////////////


     //////////////// FIM FUNÇÕES NATIVAS DO FEDERADO (GERAÇÃO AUTOMÁTICA) /////////////////////
```

```
/////////////////////////////// INICIO TRADUTOR ////////////////////////////////////

public long ToLong(String value)
{
    return Long.parseLong(value);
}
public double ToDouble (String value)
{
    return Double.parseDouble(value);
}

public float ToFloat(String value)
{
    return Float.parseFloat(value);
}

public int ToInt(String value)
{
    return Integer.parseInt(value,16);
}

public char ToChar(String value)
{
    return value.charAt(0);
}

public char[] ToCharArray(String value)
{
    return value.toCharArray();
}

public boolean ToBoolean (String value)
{
    boolean bool = false;
    String val = value.toUpperCase();

    if (val.compareTo("TRUE") == 0)
        bool = true;
    return bool;
}

public static void UpdateAttribute(String Name, String Value)
{
    App.NewEDCB.Update(Name,Value);
}

public static void UpdateAttribute(String Name, int Value[])
{
    String Temp = "";

    for (int i=0; i<Value.length; i++)
    {
        Temp += String.valueOf(Value[i]);
    }
    App.NewEDCB.Update(Name,Temp);
}

public static void UpdateAttribute(String Name, int Value)
{
    App.NewEDCB.Update(Name,Integer.toHexString(Value));
}

public void UpdateAttribute(String Name, boolean Value)
{
    App.NewEDCB.Update(Name,String.valueOf(Value));
}

public void UpdateAttribute(String Name, float Value)
{
    App.NewEDCB.Update(Name,String.valueOf(Value));
}

public void UpdateAttribute(String Name, double Value)
{
    App.NewEDCB.Update(Name,String.valueOf(Value));
}

public void UpdateAttribute(String Name, long Value)
{
    App.NewEDCB.Update(Name,String.valueOf(Value));
}

public void UpdateAttribute(String Name, char Value)
{
    App.NewEDCB.Update(Name,String.valueOf(Value));
```

```
    }

    public void UpdateAttribute(String Name, char[] Value)
    {
        App.NewEDCB.Update(Name,String.valueOf(Value));
    }
    ////////////////////////////////// FIM TRADUTOR /////////////////////////////////////////
}
```

ANEXO C EMBAIXADOR DO FEDERADO (EF)

(EF.java)

```
import java.io.*;
import java.util.*;

public class EF
{
     private int LVT;
     private ApplicationDCB App;
     public ArrayList InputRegisterList= new ArrayList();  // Lista de objetos contendo as
                                                            // informações sobre os atributos
                                                            // locais do federado (nome e tipo)
     public ArrayList InputAttributeQueue = new ArrayList();

     ////////////////////////////////////////////////////////////////////////////////////
     public EF(ApplicationDCB A) throws IOException // construtor
     {
          App = A;
          LVT = 0;
          System.out.println("EF Inicializado...");
     }

     ////////////////////////////////////////////////////////////////////////////////////
     public String getLVT()  // Retorna o LVT (Local Virtual Time)
     {
          return String.valueOf(LVT);
     }

     ////////////////////////////////////////////////////////////////////////////////////
     public synchronized void Decode(Message Msg) throws IOException // Decodifica a mensagem
     {
          InputAttribute AttributeTemp = null;
          String Source = Msg.FederationSource + Msg.FederateSource;

          App.NewEDCB.Store("INPUT",Msg.AttributeID, Msg.Value, getLVT());

          AttributeTemp = new InputAttribute(Msg.AttributeID,
                                             Msg.Value,
                                             Source,
                                             getLVT(),
                                             getAttributeType(Msg.AttributeID));

          InputAttributeQueue.add(AttributeTemp);

          App.NewGateway.ProtocolConverter(getProtocolConverterID(Msg.AttributeID));
     }

     ////////////////////////////////////////////////////////////////////////////////////
     public String getAttributeType (String uid) throws IOException
     {
          InputRegister InputRegisterTemp = null;

          for (int x=0; x < InputRegisterList.size() ; x++)
          {
               InputRegisterTemp = (InputRegister) InputRegisterList.get(x);
               if (uid.compareTo(String.valueOf(InputRegisterTemp.uid)) == 0)
                    break;
               else
                    InputRegisterTemp = null;
          }
          return InputRegisterTemp.type;
     }


     ////////////////////////////////////////////////////////////////////////////////////
     public int getProtocolConverterID(String uid)
     {
```

```
        String pc_id="";
        char tmp = ' ';

        for (int i=0; i < uid.length(); i++)
        {
            tmp = uid.charAt(i);
            if (tmp != '.')
                pc_id = pc_id + tmp;
            else
                break;
        }
        return Integer.parseInt(pc_id);
    }

    ////////////////////////////////////////////////////////////////////////////////////
    public InputAttribute getAttributeReceived(String uid)
    {
        InputAttribute Temp = null;

        for (int x = 0; x < InputAttributeQueue.size(); x++)
        {
            Temp = (InputAttribute) InputAttributeQueue.get(x);
            if (uid.compareTo(Temp.uid) == 0)
                break;
            else
                Temp = null;
        }
        return Temp;
    }

    ////////////////////////////////////////////////////////////////////////////////////
    public InputAttribute getAttributeReceived(String uid, String Source, String LVT)
    {
        InputAttribute Temp = null;

        for (int x = 0; x < InputAttributeQueue.size(); x++)
        {
            Temp = (InputAttribute) InputAttributeQueue.get(x);
            if (uid.compareTo(Temp.uid) == 0
             && Source.compareTo(Temp.Source)== 0
              && LVT.compareTo(Temp.LVT) == 0)
                    break;
            else
                Temp = null;
        }
        return Temp;
    }

    ////////////////////////////////////////////////////////////////////////////////////
    public void AttributeRemove (InputAttribute AttribRemove)
    {
        for (int x = 0; x < InputAttributeQueue.size(); x++)
            if (AttribRemove == (InputAttributeQueue.get(x)))
            {
                InputAttributeQueue.remove(x);
                break;
            }
    }
}
```

ANEXO D EMBAIXADOR DO DCB (EDCB)

(EDCB.java)

```
import java.net.*;
import java.io.*;
import java.util.*;

public class EDCB
{
     private ApplicationDCB App;
     public ArrayList OutputRegisterList = new ArrayList();  // Lista de objetos contendo as
                                                            // informações sobre os atributos
                                                            // de saída

     public ArrayList OutputAttributeQueue = new ArrayList();  // Fila de saída p/ atributos
                                                              // a serem enviados
     public boolean waiting = false;
     public boolean init = true;

     ////////////////////////////////////////////////////////////////////////////////////////
     public EDCB(ApplicationDCB A) // construtor
     {
          App = A;
          System.out.println("EDCB Inicializado...");
     }

     ////////////////////////////////////////////////////////////////////////////////////////
     public synchronized void Update(String Attribute, String Value) // Recebe atualização do Gateway
     {
          OutputAttribute OutMessage;
          OutputAttribute NextMessage;
          Destination NewDestination;

          OutputRegister NewAttribute = getOutputRegister(Attribute); // busca a instancia do objeto
                                                                      // OutputRegister que possui as
                                                                      // informações do atributo
                                                                      // + lista de destinos

          if (NewAttribute != null)
          {
               for (int x=0; x < NewAttribute.DestinationList.size(); x++)  // Monta uma mensagem para
                                                                            // cada destino diferente
               {
                    NewDestination = (Destination) NewAttribute.DestinationList.get(x);

                    OutMessage = new OutputAttribute (NewDestination.federationid,
                                                      NewDestination.federateid,
                                                      NewDestination.attributeID,
                                                      Value);

                    OutputAttributeQueue.add(OutMessage); // Adiciona a mensagem na fila de saída

                    if (init && !waiting)          // Se for a primeira mensagem a ser enviada
                    {                              // e não estiver esperando pelo envio de nenhuma
                         init = false;
                         waiting = true;
                         try
                         {
                              Code();
                         }
                         catch (IOException e) {}
                    }
               }
          }
     }

     ////////////////////////////////////////////////////////////////////////////////////////
```

```
public synchronized void SendNextMessage()
{
     try
     {
          new SendNext();

     } catch (IOException e) {}
}

///////////////////////////////////////////////////////////////////////////////////////
public class SendNext extends Thread
{
     public SendNext() throws IOException
     {
          this.start();
     }

     public void run()
     {
          if (!waiting)
          {
               try
               {
                    waiting = true;
                    init = false;

                    Code();
               }
               catch (IOException e) {}
          }
     }
}


///////////////////////////////////////////////////////////////////////////////////////
public synchronized void Code() throws IOException // construtor
{
    String Attribute;
     String Value;
     Message MessageToSend;
     OutputAttribute NewOutput;

    try
     {
          NewOutput = getFirstRequest();

          if (NewOutput != null)
          {
               MessageToSend = new Message ("Update",     App.UniqueFederationID,
                                                          App.UniqueFederateID,
                                                          NewOutput.federationid,
                                                          NewOutput.federateid,
                                                          NewOutput.attributeID,
                                                          NewOutput.Value,
                                                          App.NewEF.getLVT(),
                                                          "none",
                                                          "none");

               App.NewDCB.DCBSend(MessageToSend); // Repassa o objeto mensagem para
                                                  // a função DCBSend do DCB

               Store("OUTPUT",NewOutput.attributeID, NewOutput.Value, App.NewEF.getLVT());

          }
          else  // fila <OutputAttributeQueue> está vazia
          {
               waiting = false;
               init = true;
          }

     }
     catch (IOException e) { }
}

/////////////////////////////////////////////////////////////////////////////////////
//
//   Salva todos os atributos recebidos / enviados em um histórico
//
/////////////////////////////////////////////////////////////////////////////////////
public synchronized void Store(String StoreType, String Attribute, String Value, String LVT)
{
     FileOutputStream FileOutput;
     PrintStream File;
     String OutputString = "LVT=" + LVT + "%TYPE=" + StoreType
                        + "%ATTRIBUTE=" + Attribute + "%VALUE=" + Value;

     try
     {
```

```
            FileOutput = new FileOutputStream("history.dcb", true);
        File = new PrintStream(FileOutput);
            File.println(OutputString);
            File.close();
            FileOutput.close();
        }
        catch(IOException e) { }
    }

    ////////////////////////////////////////////////////////////////////////////////////
    //
    //   Retorna o objeto OutputRegister que possui o name passado por parametro
    //
    ////////////////////////////////////////////////////////////////////////////////////
    public OutputRegister getOutputRegister(String uid)
    {
        OutputRegister Temp = null;

        for (int x = 0; x < OutputRegisterList.size(); x++)
        {
            Temp = (OutputRegister) OutputRegisterList.get(x);

            if (uid.compareTo(Temp.uid) == 0)  // compara se as strings são iguais
                break;
            else
                Temp = null;
        }
        return Temp;
    }


    ////////////////////////////////////////////////////////////////////////////////////
    public synchronized OutputAttribute getFirstRequest()
    {
        OutputAttribute Temp = null;
        int index = -1;

        if (OutputAttributeQueue.size() > 0)
        {
            Temp = (OutputAttribute) OutputAttributeQueue.get(0);
            OutputAttributeQueue.remove(0);
        }

        return Temp;
    }
}
```

ANEXO E ARQUIVO DE CONFIGURAÇÃO PARA O FEDERADO DISPLAY NO EXPERIMENTO 2

(config.xml)

```
<?xml version="1.0" encoding="UTF-8"?>

<CONFIG>
  <INFO federateid="1" federationid="1" localport="4000" type="synchronous">
    <ATTRIBUTE id="1.1" name="address.pos" type="int" />
    <ATTRIBUTE id="2.0" name="clean" type="function" />
    <ATTRIBUTE id="3.1" name="writeInt.value" type="int" />
    <ATTRIBUTE id="4.1" name="writeChar.asciiCode" type="int" />
    <ATTRIBUTE id="5.0" name="initialize" type="function" />
  </INFO>
</CONFIG>
```

ANEXO F MODELO DA FEDERAÇÃO (XML) UTILIZADO NO EXPERIMENTO 1

(federation.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<CONFIG>

<FEDERATION id="1">

<FEDERATE id="1" ip="127.0.0.1" port="4000" type="synchronous" code_location="C:\display"
interface_type="java">
   <FUNCTION uid="1.0" name="send" return_type="void">
       <PARAMETER uid="1.1" name="message" type="String"/>
   </FUNCTION>
</FEDERATE>

<FEDERATE id="2" ip="127.0.0.1" port="4001" type="synchronous" code_location="C:\teclado"
interface_type="java">
   <ATTRIBUTE uid="1.0" name="value" port_type="OUT" type="int">
          <DESTINATION federationid="1" federateid="3" attribute="3.1"/>
   </ATTRIBUTE>
</FEDERATE>

<FEDERATE id="3" ip="127.0.0.1" port="4002" type="synchronous" code_location="C:\gpsalerta"
interface_type="java">
   <ATTRIBUTE uid="1.0" name="message" port_type="OUT" type="String">
       <DESTINATION federationid="1" federateid="1" attribute="1.1"/>
   </ATTRIBUTE>
   <FUNCTION uid="2.0" name="receive" return_type="void">
       <PARAMETER uid="2.1" name="pos" type="String"/>
   </FUNCTION>
   <FUNCTION uid="3.0" name="send" return_type="void">
       <PARAMETER uid="3.1" name="value" type="int"/>
   <ATTRIBUTE uid="4.0" name="send" port_type="OUT" type="String">
       <DESTINATION federationid="1" federateid="4" attribute="2.0"/>
   </ATTRIBUTE>
</FEDERATE>

<FEDERATE id="4" ip="127.0.0.1" port="4003" type="synchronous" code_location="C:\gps"
interface_type="java">
   <ATTRIBUTE uid="1.0" name="pos" port_type="OUT" type="String">
        <DESTINATION federationid="1" federateid="3" attribute="2.1"/>
   </ATTRIBUTE>
   <FUNCTION uid="2.0" name="Send" return_type="void">
   </FUNCTION>
</FEDERATE>

</FEDERATION>

</CONFIG>
```