DESENVOLVIMENTO DE UM AMBIENTE DE CO-SIMULAÇÃO DISTRIBUÍDO E HETEROGÊNEO

Alexandre Amory, Fernando Moraes, Leandro Oliveira, Fabiano Hessel [1], Ney Calazans
{ amory, moraes, laugusto, hessel, calazans}@inf.pucrs.br

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Faculdade de Informática
Av. Ipiranga, 6681 - Prédio 30 / BLOCO 4
Telefone: +55 51 3320-3611 - Fax: +55 51 3320-3621
CEP 90619-900 - Porto Alegre - RS – BRASIL

ABSTRACT

This paper presents the implementation of a hardware and software co-simulation environment. Different simulators, which can be geographically distributed, compose this environment. The communication between simulators (gcc, ModelSim) is done using Unix sockets, through a router program. The router reads a file containing the communication between modules, launches the simulators and controls the simulation synchronization. A simple case study is presented, to validate this co-simulation methodology.

RESUMO

Este trabalho apresenta a implementação de um ambiente de simulação conjunta entre módulos hardware e software. Este ambiente é composto por diversos simuladores para cada módulo do sistema, que podem estar distribuídos em uma WAN (*wide area network*). Os simuladores se comunicam com uso de *sockets* através de um programa roteador que estabelece a conexão entre todos os módulos. Um simples estudo de caso é apresentado com o objetivo de validar esta metodologia de co-simulação.

Palavras-chave: co-simulação, projeto conjunto *hardware*/*software*, co-design.

1 INTRODUÇÃO

A atual complexidade dos sistemas computacionais é tal que o projeto não pode mais ser realizado com uma única linguagem nem com um único nível de abstração [7]. Atualmente sistemas computacionais podem ser compostos de módulos de *hardware*, *software*, e podem ainda conter módulos analógicos e partes mecânicas [5]. Desta forma, diferentes linguagens e modelos de computação são utilizados para cada um destes domínios. Na concepção de sistemas heterogêneos complexos, seus módulos são desenvolvidos e validados em separado. Porém, em todas etapas do projeto é necessária uma validação total do sistema. A validação deste tipo de sistema é uma tarefa complexa devido a sua heterogeneidade. Entretanto, a técnica de validação mais utilizada é simulação, uma vez que as técnicas de verificação formal ainda são imaturas [1]. O princípio da co-simulação é a execução paralela dos simuladores necessários para a validação de um sistema. Cada simulador é responsável por executar um módulo do sistema em uma linguagem, modelo de computação e nível de abstração específico ao domínio tratado [3]. A Figura 8, apresentada no fim do artigo, situa a etapa de validação por co-simulação no contexto do fluxo de *co-design* que está sendo desenvolvido. A cada transição entre níveis de abstração ocorre uma validação por meio de co-simulação.

O propósito deste trabalho é o desenvolvimento de um ambiente de co-simulação para módulos de hardware e software que suporte simulação distribuída. Desta forma, este trabalho esta organizado da seguinte forma. A Seção 2 apresenta uma breve distinção entre modelagem homogênea e heterogênea de sistemas computacionais complexos. Na Seção 3 são discutidas uma série de características de alguns ambientes de co-simulação descritos na literatura e decisões de projeto que devem ser tomadas durante o desenvolvimento de um ambiente de co-simulação. Na Seção 4 é apresentado o ambiente de co-simulação proposto e na Seção 5 apresentamos as conclusões e trabalhos futuros.

2 MODELAGEM DE SISTEMAS

A presença da grande variedade de modelos de computação empregados em sistema heterogêneo é devido as diferentes funcionalidades envolvidas em um sistema. Por exemplo, para descrever um sistema composto de *hardware* e *software*

[1] Financiado pelo projeto CNPq 68.0117/01-6 e pela Fapergs, projeto 00/2227.7

poderíamos escolher as linguagens VHDL e C, respectivamente, para sua especificação. Porém, se a decisão de projeto for adotar uma *única linguagem* (*modelagem homogênea*) de especificação, teríamos C descrevendo *hardware* e, naturalmente, *software* ou VHDL descrevendo *software* e *hardware*. Esta abordagem teria a grande vantagem de necessitar de um único conjunto de ferramentas para validar todo o sistema [9]. É evidente que a escolha de uma única linguagem apresenta limitações devido ao fato do modelo de computação empregado não ser aplicado aos dois domínios (*hardware* e *software*). Apesar disto, algumas abordagens tentam diminuir estas limitações adicionando bibliotecas que incluem funcionalidades que implementam diferentes modelos de computação [10]. Outras abordagens tentam definir uma única linguagem que possua todos os conceitos necessários para a especificação de um sistema, porém é difícil definir tal linguagem [2] [6].

A alternativa defendida é manter as diferenças conceituais dos diferentes domínios. Isto implica em modelar cada módulo do sistema em uma linguagem específica e apropriada (*modelo heterogêneo multi-linguagem*). Porém o problema principal no uso desta abordagem é definir a semântica de interação de modelos computacionais diferentes e uma técnica de sincronização entre os processos (simuladores) [1]. Desta forma linguagens como SDL e C++ podem der utilizadas em descrições mais abstratas, matlab pode ser usado para modelar partes mecânicas e processos físicos, VHDL-AMS pode ser utilizado para descrição de *hardware* analógico, C e assembly para descrever *software* e VHDL para descrever *hardware* digital.

3 CARACTERÍSTICAS DE UM AMBIENTE DE CO-SIMULAÇÃO

Nesta Seção apresentamos as principais características e decisões de projeto de um ambiente de co-simulação de acordo com a literatura.

- **Motor de Simulação:**

A *Co-simulação mono-motor* consiste em transformar os módulos descritos em diversas linguagens em uma mesma representação que descreve todo o sistema [3]. A simulação ocorre nesta representação única do sistema, também chamada de formato intermediário. A vantagem desta abordagem é a necessidade de uma única ferramenta de simulação para validar todo o sistema. Porém tem a desvantagem de ser difícil converter qualquer linguagem com qualquer modelo de computação em um formato intermediário, pois seria difícil desenvolver um formato intermediário que suporte tais variações.

Na *co-simulação multi-motor* cada módulo é simulado por um simulador apropriado para cada linguagem [3].

A co-simulação se resume a uma troca de mensagens entre os simuladores. A vantagem deste método é que permite a utilização das ferramentas existentes para realizar a simulação. Porém, é necessário que os simuladores envolvidos permitam o envio de seus dados de simulação para o exterior, via uma interface de comunicação. Além disto surge a necessidade da utilização de um método de sincronismo para coordenar a troca de dados entre esses simuladores que estão executando paralelamente ou concorrentemente. A Figura 1 ilustra estes conceitos.

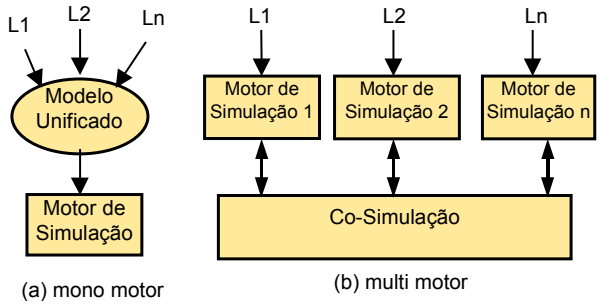

Figura 1 – Motor de Simulação.

- **Mecanismos de Comunicação:**

Se for adotado o conceito de múltiplos motores de simulação, tem-se que escolher algum mecanismo de comunicação entre os diversos simuladores. A escolha do mecanismo de comunicação é feita através de um compromisso entre desempenho, flexibilidade e suporte a co-simulação distribuída. Mecanismos de comunicação entre processos como *pipes*, por exemplo, não suportam comunicação distribuída. Por outro lado *sockets* além de suportar comunicação distribuída, é suportada por diversos sistemas operacionais e linguagens de programação. Já mecanismos como *corba* e *RMI* conferem ao sistema uma maior flexibilidade, abstração e suporte a simulação distribuída.

- **Modelo Temporal:**

A *validação funcional* consiste em validar a funcionalidade do sistema sem sincronização temporal dos simuladores durante a execução da co-simulação [3]. A troca de dados entre simuladores é controlada por eventos. A validação funcional é utilizada em níveis de abstrações elevados onde a noção de tempo não é necessária à simulação.

A *validação temporal* permite realizar uma validação mais realista do sistema, considerando o tempo [3]. O envio e a recepção de dados acontece em instantes precisos de tempo, chamados de janelas temporais. Assim, os diferentes simuladores são sincronizados e todos possuem o mesmo

relógio global de simulação. Quando um dado não é consumido em um intervalo específico de tempo, este dado pode ser perdido. Este tipo de validação é muito utilizado para a validação de um sistema especificado em um nível de abstração baixo, onde torna-se necessário realizar uma simulação precisa.

- **Modelo de Sincronização:**

Ao se optar pelo desenvolvimento de um ambiente que suporte co-simulação temporal deve-se definir qual é o modelo de sincronismo utilizado, já que os diversos simuladores que compõem o sistema podem ter velocidades de execução e modelos de tempo diferentes. A função básica do modelo de sincronização é coordenar o envio e recepção de dados através do ambiente de co-simulação. A Figura 2 ilustra a diferença destas duas opções de sincronização.

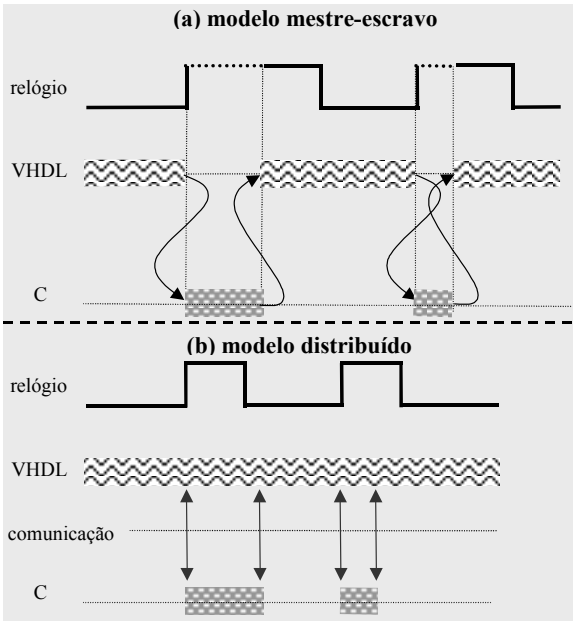

Figura 2 – Modelo de Sincronização [6].

Existem diversas abordagens para sincronização. A mais simples é o *modelo mestre/escravo* que consiste em designar um simulador como mestre da simulação, o qual fica responsável por invocar os outros simuladores escravos [3]. A maior vantagem deste modelo é a simplicidade. Porém, este modelo restringe a arquitetura do sistema aos sistemas mestres/escravos, formados por um processador mestre ao qual são associados coprocessadores escravos. Este modelo de sincronização elimina toda forma de paralelismo entre os módulos.

O *modelo distribuído* permite a execução concorrente dos simuladores. Cada simulador pode enviar e receber dados sem restrições. Este modelo apresenta a vantagem de não restringir a arquitetura do sistema e assim permitir a sua utilização em um grande número de aplicações [3] [6]. No entanto, a coerência entre os dados compartilhados pelos simuladores torna-se mais difícil de assegurar. A principal metodologia empregada para este modelo consiste em conectar cada simulador em um barramento de co-simulação, o qual é encarregado do envio e recepção de dados, e da sincronização entre os simuladores. Este barramento age como um servidor de comunicação e sua implementação pode ser baseada em mecanismos de comunicação entre processos, como por exemplo ipc ou *sockets* no caso do sistema UNIX.

3.1 Decisões de projeto

Alguns dos principais *desafios encontrados no desenvolvimento de um ambiente de co-simulação* são sincronização, conversão de dados, desempenho de comunicação (tempo de simulação), facilidade de integração de novos simuladores e compromisso entre desempenho e precisão.

- **Sincronização:**

Quando se opta por desenvolver um ambiente de co-simulação distribuído com co-simulação temporal deve-se escolher o método de sincronização entre os simuladores. Diversos algoritmos diferentes são implementados na literatura. Algumas alternativas são o algoritmo "*synchronized handshake*" [4] e "*lock step*" [5].

- **Conversão de dados:**

Uma vez que o sistema é composto por diversos modelos de computação que podem conter tipos de dados diversos, é necessário definir uma estratégia responsável pela conversão de dados. Na Seção 4.2 apresentamos a solução de conversão de dados implementada.

- **Desempenho de comunicação:**

A co-simulação distribuída é a capacidade de simular um projeto em máquinas distintas geograficamente distribuídas sobre o suporte de uma LAN (*local area network*) ou WAN (*wide area network*). Algumas das motivações para a pesquisa neste campo são: descentralização do projeto, facilidade de cooperação entre diferentes grupos do projeto, gerenciamento de propriedade intelectual, gerenciamento de licenças de simuladores e compartilhamento de recursos. Porém, a desvantagem desta abordagem é que o desempenho da simulação se torna dependente do desempenho da rede. Pelo fato de permitir a execução paralela e concorrente de diversos simuladores, outro problema que surge é a necessidade de sincronização destes simuladores.

- **Facilidade de integração de novos simuladores:**

A abordagem tradicional para integração de novos simuladores é baseada na construção caso a caso das interfaces entre o ambiente de co-simulação e o simulador [7]. Quando um novo simulador for inserido no sistema, uma nova interface deve ser desenvolvida com pouco ou nenhum reuso de código, uma vez que na maioria dos casos as funções do simulador que provêm comunicação com o mundo externo são proprietárias. Deste ponto de vista, a maioria das abordagens são adequadas para atender um conjunto de problemas pré definidos. Porém, problemas inesperados podem exigir a integração de novos simuladores. O desenvolvimento de uma nova interface pode ser inaceitável por questão de tempo ou por impor restrições ao simulador para se adaptar ao ambiente de co-simulação. Estas restrições poderiam comprometer, por exemplo, a precisão da simulação.

Porém, em [7] é apresentada uma abordagem mais flexível que as abordagens atuais. Este utiliza um padrão do IEEE, chamado HLA (*high level architecture*), que propõe regras e mecanismos para facilitar interoperabilidade de simuladores heterogêneos distribuídos. Desta forma, todo o simulador que reconheça este padrão pode ser adicionado ao sistema de forma transparente.

- **Compromisso entre desempenho e precisão:**

A maioria das abordagens que trata este item permite que seja possível escolher o nível de abstração em que cada módulo é simulado. Desta forma, os módulos que podem ser simulados com menor número de detalhes contribuem com o desempenho de simulação do sistema uma vez que diminui a quantidade de comunicação [5]. Inicialmente, no nível de abstração sistêmico, as primitivas de comunicação entre os módulos são modeladas como uma única chamada a primitivas *send* e *receive*, sem qualquer preocupação com o protocolo de comunicação entre estes módulos. Com o aumento de detalhes de projeto, as primitivas também são detalhadas de forma a especificar um protocolo de comunicação. Por exemplo, a comunicação realizada no nível sistêmico, que utilizava somente uma chamada de *send* ou *receive*, no nível de implementação (arquitetural) é desmembrada em diversas chamadas de comunicação que implementam um protocolo, como por exemplo *handshake*. Desta forma, de uma única chamada de comunicação no nível sistêmico, existirão três chamadas de comunicação (*request*, *data* e *ack*) para o nível arquitetural.

Porém em [4] é apresentada uma abordagem diferenciada, chamada técnica de prototipação gradual, que tem como objetivo diminuir o tempo total de simulação (desempenho) prototipando em FPGA componentes de *hardware* já validados e sintetizados. A cada módulo de *hardware* prototipado, menor fica o sistema que está sendo simulado, diminuindo a complexidade e o tempo de simulação . Este processo de validação por prototipação gradual continua até que todos módulos de *hardware* tenham sido prototipados. Desta forma, ao fim da validação do sistema, os componentes de *hardware* já estão prototipados. A co-simulação com FPGA obteve um ganho de desempenho de 1.7 sobre a co-simulação baseada puramente em *software* [4].

4 AMBIENTE DE CO-SIMULAÇÃO PROPOSTO

Um ambiente de co-simulação distribuído é apresentado, de forma geral, como a estrutura mostrada na Figura 3. Para simular a existência de um barramento de interconexão, utiliza-se primitivas de *sockets* para comunicação entre os módulos e um roteador de mensagens, que recebe as mensagens enviadas pelos módulos e as envia para todos os módulos que devam tomar conhecimento de tal mensagem. Esta organização está descrita na Figura 3.

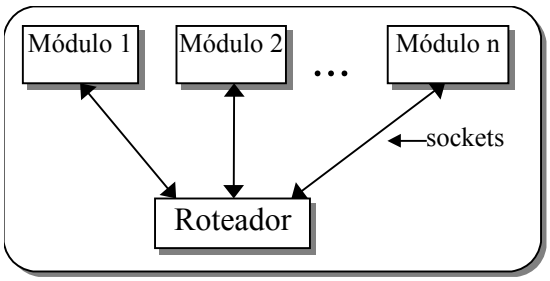

Figura 3 – Arquitetura do Ambiente de Co-simulação

Para comunicar-se com o ambiente de co-simulação, cada módulo deve utilizar uma biblioteca especificamente desenvolvida para seu simulador. Estas bibliotecas são partes integrantes do ambiente de co-simulação. Módulos escritos em VHDL utilizarão entidades para envio e recebimento de dados. Maiores descrições das bibliotecas de comunicação são encontradas na Seção 4.2.

A linguagem VHDL não possui interface com *sockets*. No entanto, alguns simuladores permitem que se defina o comportamento de uma entidade VHDL em código C, utilizando uma biblioteca. Ferramentas de simulação VHDL como ModelSim e Synopsys possuem bibliotecas chamadas FLI (*Foreign Language Interface*)[8] e CLI (*C Language Interface*), respectivamente. Desta forma é possível utilizar a biblioteca de *sockets* do sistema operacional para realizar a comunicação com o ambiente de co-simulação.

4.1 Estrutura do Roteador

A função principal do roteador é coordenar a comunicação entre os diversos módulos. Para que isso seja possível, o roteador necessita conhecer as definições dos módulos e suas interconexões. Para este fim, é fornecido um arquivo de coordenação, que é descreve o nível mais alto de hierarquia do projeto. Um exemplo deste arquivo é apresentado na Figura 4.

```
1.  module VHDL {
2.  language("VHDL");
3.  simulator("ModelSim");
4.  machine(name="hios.inf.pucrs.br",alias="hios");
5.  port {   int A : in;
6.           int B : in;
7.           int comeca : in;
8.           int result : out;}
9.  }
10. module M2 {
11. language("C");
12. simulator("ddd");
13. machine(name="wacken.inf.pucrs.br",alias="wacken");
14. port {   int A : out;
15.          int B : out;
16.          int comeca : out;
17.          int result : in;}
18. }
19. net N1 { A(VHDL),A(M2) }
20. net N2 { B(VHDL),B(M2) }
21. net N3 { result(VHDL),result(M2) }
22. net N4 { comeca(VHDL),comeca(M2) }
```

Figura 4 – Arquivo de Coordenação.

Neste exemplo, a co-simulação é realizada entre dois módulos, chamados *VHDL* e *M2*. O módulo *VHDL* é especificado na linguagem VHDL e simulado com a ferramenta ModelSim e a biblioteca FLI. Este módulo é executado na máquina especificada pela diretiva *machine*, e possui as portas *a*, *b*, *comeca* e *result* do tipo inteiro. O módulo *M2* é descrito em linguagem C, tendo as portas necessárias para a comunicação com o módulo *VHDL*. Este arquivo de coordenação é utilizado no estudo de caso da Seção 4.3.

A seguir temos as definições de interconexões entre os módulos, representados pela diretiva *net*. Cada interconexão tem um nome (no exemplo acima, N1 a N4). Entre chaves são especificadas as portas que fazem parte da interconexão, no formato porta (módulo).

A partir do arquivo de coordenação o roteador monta suas estruturas internas de controle da co-simulação, tornando possível o roteamento de mensagens entre os diversos módulos participantes da co-simulação.

A mensagem que trafega entre os módulos traz basicamente a informação sobre qual porta de qual modulo acionou o envio da mensagem e qual o novo valor da porta.

A co-simulação é interrompida no momento em que algum dos módulos se desconecta do sistema. Entretanto, o sistema garante que todas as mensagens já enviadas serão recebidas pelos módulos que ainda estiverem conectados.

Das classes que compõem o sistema, as seguintes são as mais importantes:

- **CCoSim**: É a interface para todo o sistema. Suas principais funções são: carregar o arquivo de coordenação, montando as estruturas internas do roteador; receber as conexões de todos os clientes, fazendo a autenticação dos mesmos; e rotear as mensagens que chegam para os módulos que tem portas conectadas a mesma rede da porta do modulo que originou a mensagem;
- **CPort**: Representa uma porta de um módulo. Tem como atributos o tipo da porta e sua direção (que pode ser de entrada, saída ou entrada e saída);
- **CNet:** Representa uma conexão entre as portas dos módulos;
- **CModule:** Representa um módulo, contendo informações sobre suas portas e o *socket* que está sendo usado para se comunicar com o simulador do módulo.

Para a execução do ambiente é necessário que cada módulo saiba em qual endereço (IP + porta) se localiza o roteador. No momento da conexão o roteador verifica se o módulo faz parte da co-simulação através das definições fornecidas pelo arquivo de coordenação. Isto torna necessário que o módulo tenha conhecimento do seu nome para conseguir se autenticar junto ao roteador.

A Figura 9, no fim do artigo, mostra o diagrama de classes do ambiente. A classe principal do roteador é CcoSim, através dela é que são realizadas as operações de leitura do arquivo de coordenação, geração da estrutura de conexão dos módulos e o roteamento das mensagens. Como pode ser visto no diagrama, a classe tem conexão com módulos (CModule) e redes (CNet). Desta forma pode-se saber quais módulos fazem parte da co-simulação, e quais são as interconexões entre estes módulos. As redes sabem quais portas (Cport) estão ligadas a ela, podendo saber qual módulo está ligado àquela porta.

No momento que o roteador recebe uma mensagem, ele verifica em qual *net* a porta originadora está conectada, e procede enviando mensagens para todos os módulos que têm portas conectadas a esta *net*, através do método SendMsg da classe CModule. A classe CModule guarda informação sobre qual *socket* mantêm a conexão

com o simulador do módulo correspondente, podendo assim enviar a mensagem ao mesmo.

4.2 Bibliotecas de Comunicação

A biblioteca de comunicação tem a função de integrar um simulador ao ambiente de co-simulação. Esta biblioteca consiste basicamente de três funções: inicialização (csiInitialize( )), envio de dados (csiSend( )) e recebimento de dados (csiReceive( )).

A Figura 5 apresenta os módulos principais que compõem o ambiente de co-simulação, incluindo o projeto do usuário. Os módulos em cor clara representam partes do ambiente de co-simulação, os outros dois módulos compõem o projeto do usuário. Para utilizar o ambiente de co-simulação o usuário deve incluir em cada módulo a ser simulado a biblioteca de comunicação com o ambiente de co-simulação. Na Figura 5 estas bibliotecas são designadas *libCom*. Cada simulador deve possuir a sua biblioteca de comunicação, pois geralmente a interface do simulador com o mundo externo é proprietária. Atualmente o sistema possui bibliotecas que suportam C e VHDL. Na parte de software do projeto usuário, o usuário deve mudar manualmente o código incluindo a primitiva de inicialização, que conecta o simulador no ambiente, e as primitivas *send* e *receive*.

Na Figura 7 da Seção 4.3 apresentamos o código fonte do usuário já com as chamadas apropriadas para integrar o software ao sistema. Já na parte de hardware, o usuário deve mudar, também manualmente, a descrição do seu projeto de forma a adicionar um componente VHDL que instancia o módulo do usuário. Além disto, este componente contém componentes de comunicação (ver na Figura 5) que possibilitam a troca de mensagens entre as portas do hardware e do software.

Na Figura 10, no fim do artigo, da Seção 4.3 apresentamos a estrutura deste *testbench* para o estudo de caso do multiplicador por somas parciais. Alterado o projeto do usuário, deve ser criado o arquivo de coordenação que conecta os módulos do usuário ao ambiente. Um exemplo deste arquivo pode ser encontrado na Figura 4. Feito isto, o ambiente pode ser inicializado. Primeiro inicializa-se o roteador, que começa aguardando a conexão de todos os módulos indicados no arquivo de coordenação para iniciar a etapa de roteamento. Após, todos os simuladores que compõem o ambiente devem ser executados. Ao se conectarem, o roteador está pronto para rotear as mensagens para começar a simulação.

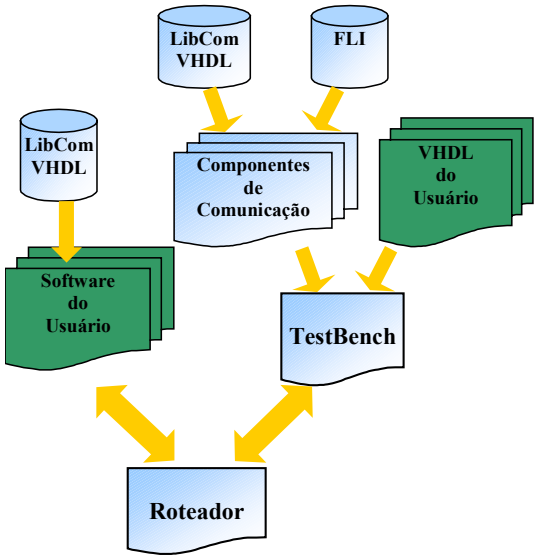

Figura 5 – Estrutura Geral do Sistema de Co-simulação em Relação a Bibliotecas Utilizadas.

Detalhando mais o funcionamento da primitiva de inicialização, o usuário deve incluir, no início do código, a função csiInitialize ( ). Esta função é responsável pela conexão do simulador no ambiente. Os parâmetros passados na sua execução são o nome do módulo do usuário e o endereço IP do roteador de mensagens do ambiente de co-simulação. Cada simulador deve informar o seu nome (nome do módulo do usuário) para o roteador, pois este deve relacionar o nome do módulo à conexão do *socket* e verificar se o nome do simulador consta no arquivo de coordenação .

No caso da utilização de VHDL com o simulador ModelSim, utilizamos uma entidade VHDL com a arquitetura definida em C através da biblioteca FLI. Assim, esta entidade contém a biblioteca FLI, que comunica com o simulador, e a biblioteca que faz a comunicação do simulador com o ambiente de co-simulação. Esta entidade, chamada de componente de comunicação, está representada na Figura 5.

Uma funcionalidade atribuída às bibliotecas de comunicação é a conversão de dados. Três alternativas foram analisadas em relação ao local onde esta conversão deve ser executada: no módulo de origem da mensagem, no roteador e nos módulos de destino. É importante lembrar que uma mesma mensagem pode ser enviada para vários módulos que podem representar diferentes simuladores e linguagens. Desta forma, não é possível que a conversão de dados ocorra no envio da mensagem, uma vez que os destinatários da mesma podem ser representados por tipos de dados diferentes. Também não seria recomendado executar esta conversão de dados no roteador, pois o roteador deveria conhecer todos os tipos de dados de todos

os simuladores do sistema. Assim, sempre que um novo simulador for adicionado, o roteador deveria ser recompilado de forma a incluir a biblioteca de conversões de dados deste novo simulador. Desta forma, a melhor opção de onde executar a conversão de dados é no destino da mensagem, pois torna o ambiente mais escalável e sem necessidade de recompilação. É importante destacar que esta opção implica na escolha de um tipo de dado único que trafega no sistema. Este tipo de dado deve ser compatível com todos os simuladores que compõem o sistema.

4.3 Estudo de caso

Utilizamos como estudo de caso uma aplicação simples de multiplicação por somas sucessivas particionada manualmente em *software* (C) e em *hardware* (VHDL). O módulo C gera operandos e recebe os resultados das multiplicações. O módulo *hardware* executa a multiplicação propriamente dita. É importante destacar que o objetivo desta aplicação é o de validar a metodologia de co-simulação, e não de validar um sistema complexo.

Do ponto de vista do usuário, é necessário acrescentar as bibliotecas de comunicação nos módulos que participam da co-simulação. Em relação ao módulo de *hardware*, deve ser construído um *testbench* instanciando o módulo VHDL (UUT) e um componente de comunicação para cada um dos sinais (operado A, operando B, executa e resultado) que se comunicam com o *software*. A Figura 10 ilustra esta estrutura. Apesar de quatro componentes de comunicação serem instanciados, o sistema está escrito de maneira que apenas um canal de comunicação (*socket*) é aberto para se comunicar com o ambiente de co-simulação. As portas do componente de comunicação de VHDL são apresentadas na Figura 6. As portas *porta* e *envia* são constantes que definem, respectivamente, qual porta do VHDL (a, b, resultado ou começa) que esta instância da biblioteca monitora e qual o sentido da comunicação desta porta (entrada ou saída). Se a porta *envia* for definida como valor 1, então esta instância da biblioteca de comunicação somente envia dados. Desta forma, a porta *dout* fica sendo inutilizada. No caso contrário, porta *envia* com valor zero, a instância somente recebe mensagens e a porta *din* é inutilizada recebendo uma constante com zeros. A porta *executa* indica ao módulo o momento de executar um recebimento ou um envio de mensagem. Quando for executado um recebimento de mensagem, por utilizar *sockets* em modo conectado, o simulador VHDL não vai avançar no tempo de simulação, pois vai esperar até que a mensagem desejada seja recebida. Existem duas máquina de estado no *testbench* da Figura 10. A máquina de estado FSM 1 seqüencializa a ordem de ativação da porta *executa* dos componentes de recepção de dados. A FSM 2 dispara um envio de mensagem quando a UUT indicar que o resultado da multiplicação está pronta através da porta *pronto*. Na Figura 10 também é destacada a estrutura interna do componente de comunicação no caso de ser configurado com componente de recebimento e de envio. A sigla S.O. significa a chamada ao sistema operacional realizada para executar a comunicação por *sockets*.

```
1. entity libCom is
2. generic(porta : in string;
3.         envia  : in std_logic);
4. port ( din     : in std_logic_vector(31 downto 0);
5.         dout    : out std_logic_vector(31 downto 0);
6.         executa         : in std_logic;
7.         clk     : in std_logic;
8.         reset   : in std_logic );
9. end libCom;
```

Figura 6 – Entidade do Componente de Comunicação de VHDL.

Porém, é importante relatar os problemas associados ao uso desta estrutura de *testbench* para os módulos de *hardware*. Devido às dificuldades na utilização de procedimentos VHDL com comportamento definido através de FLI, utilizamos apenas entidades, o que acabou por tornar a construção do *testbench* mais complexa. Com isto, os *testbenchs* utilizados com o sistema tendem a não ser genéricos, dificultando a utilização do ambiente pelo usuário. Outra dificuldade é que cada porta de comunicação do sistema pode ter tipos diferentes. Com entidades, é necessário descrever vários componentes com tipos diferentes na portas de entrada e saída de dados, uma vez que não é possível em VHDL descrever um tipo de dados no campo *generic*. Se por outro lado, tivéssemos conseguido utilizar procedimentos e funções em VHDL e FLI, a modificação dos módulos *hardware* seria parecida com a modificação nos módulos de *software*, facilitando a utilização do usuário.

Em relação ao módulo de *software*, as modificações necessárias são mais simples do que os módulos de *hardware*. Na Figura 7 apresentamos o código fonte da parte de *software* deste estudo de caso. Dentre as modificações necessárias estão a inclusão da biblioteca de comunicação da linguagem C (linha 1), a inicialização da comunicação (linha 5) e as chamadas de envio e recepção de dados (linhas 8-12).

```
1.   #include "csi.h"
2.   #include <stdio.h>
3.   void main(int argc, char *argv[])
4.   {  int x=2, result;
5.      CSI csiInitialize(CSI_INIT);
6.
7.     while(1) {
8.       csi.Send("B",x);
9.       csi.Send("A",x);
10.      csi.Send("comeca",1);
11.
12.      csi.Receive("result",result);
13.      printf(" %d x %d = %d\n",x,x,result);
14.      x++;}
15.    getchar();
16.  }
```

Figura 7 – Código Fonte do *Software* de Multiplicação por Somas Parciais.

A Figura 11, no fim do artigo, apresenta a captura de tela da co-simulação do estudo de caso. Existem quatro janelas nesta figura. A janela 1 é o console do simulador VHDL. Na área destacada desta janela apresentamos o momento em que este simulador recebe os três dados (a, b e começa nesta ordem) e envia o resultado da multiplicação (196). A janela 2 representa a forma de onda da descrição de *hardware*. No momento destacado, o módulo de *hardware* sinaliza na porta *pronto* o momento que a multiplicação está realizada. O resultado da multiplicação está na porta *result*. Outros sinais desta descrição de *hardware* também são apresentados. A janela 3 representa o módulo de *software*. Cada linha impressa é o resultado de um multiplicação. Está destacada nesta janela o momento que ocorre a multiplicação 14 x 14. Por fim, a janela 4 representa o roteador. Destacamos nesta janela o momento que ocorre a mesma multiplicação (14 x 14). Note que é impresso o momento que o roteador recebe o dado 14 da porta *a* do módulo M2 (módulo de *software*). Também é apresentado o momento que o módulo chamado VHDL envia ao roteador, pela porta *result*, o valor 196.

5 CONCLUSÕES E TRABALHOS FUTUROS

Neste artigo apresentamos uma proposta de metodologia de co-simulação de sistemas heterogêneos composto por módulos de *hardware* e *software*. Este ambiente proposto suporta co-simulação geograficamente distribuída e pode ser aplicado na validação de sistemas embarcados. A metodologia foi validada através de um exemplo simples. Porém, como parte dos trabalhos futuros, avaliaremos este ambiente com sistemas mais complexos com aplicações realistas como aplicações de multimídia, imagem, entre outras. No futuro também acrescentaremos suporte a outras linguagens como SDL, para especificação sistêmica. O simulador do processador ARM também será suportado.

BIBLIOGRAFIA

[1] EDWARDS, S.; LAVAGNO, L.; LEE, E.; SANGIOVANNI-VINCENTELLI, A. "**Design of Embedded Systems: Formal Models, Validation and Synthesis**". Proceedings of the IEEE, 1997.

[2] GAJSKI, D.D.; et al. "**SpecC: Specification Language and Methodology**". Kluwer Academic. Boston, USA. 2000.

[3] HESSEL, F. "**Concepção de Sistemas Heterogêneos Multi-Linguagens**". Jornada de Atualização em Informática – JAI. XXI Congresso da Sociedade Brasileira de Computação. 2001.

[4] KIM, Y.; et. al. "**An Integrated Hardware-Software Cosimulation Environment for Heterogeneous Systems Prototyping**". Proc. Asia and South Pacific Design Automation Conference (ASPDAC), p. 101-106, 1995. http://citeseer.nj.nec.com/kim96integrated.html.

[5] LEMARREC, P.; et. al. "**Hardware, Software and Mechanical Cosimulation for Automotive Applications**". In: Proc. of International Workshop on Rapid System Prototyping (RSP'98), Leuven, Belgium, pp.202-206, 1998.

[6] MARCON, C.A.M. "Modelos, Ferramentas e Métodos para o Projeto Integrado de *Hardware* e *Software"*. Trabalho Individual I. PPGCC – PUCRS. Agosto 2000.

[7] MELLO, B.A.; WAGNER, F.R. "**A Standard Co-Simulation Backbone**". Accepted for presentation and publication in the proceedings of VLSI-SOC, to be held in Montpellier, France, December 2001. http://www.urisan.tche.br/~bmello/trab-pub.htm

[8] MODELTECH inc. "**ModelSim SE/EE user's manual**". Capítulo 12 -VHDL Foreign Language Interface (FLI). available at homepage: http://www.model.com. Version 5.4. April, 2000.

[9] OYAMADA, M.; WAGNER, F.R. "**Co-simulation of Embedded Electronic Systems**". In: ESS'2000 - 12nd. European Simulation Symposium. Hamburg, Germany, September 2000. Proceedings, Society for Computer Simulation, 2000.

[10] SWAN, S. "**An Introduction to System Level Modeling in SystemC 2.0**". white paper. Available at homepage: http://www.systemc.org. 2001

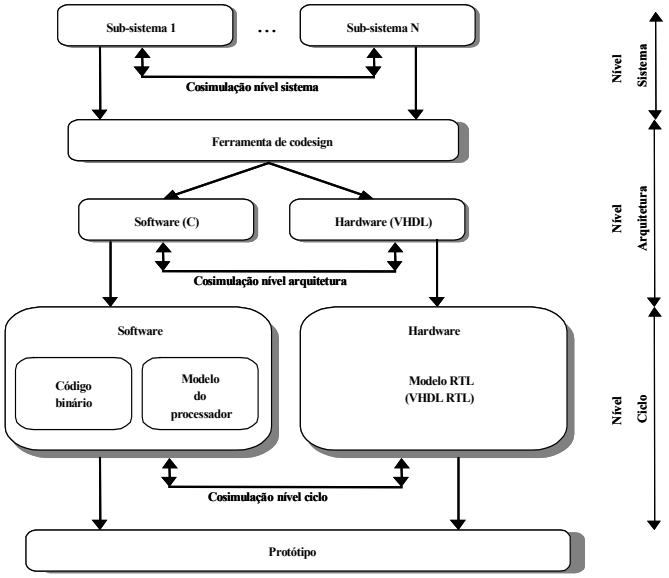

Figura 8 – Representação de um Fluxo de Co-design Destacando a Localização da Etapa de Co-simulação nos Diversos Níveis de Abstração [3].

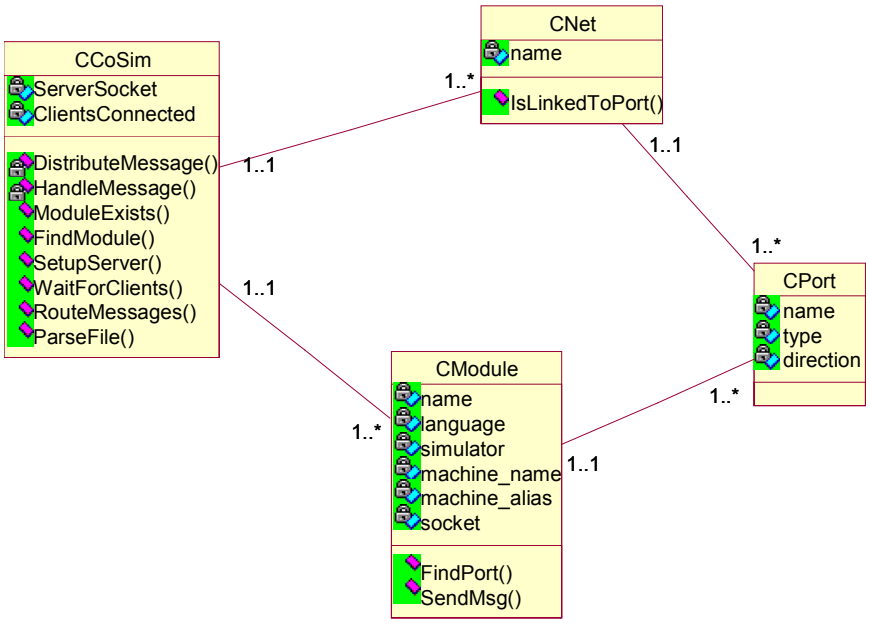

Figura 9 – Diagrama de Classes do Roteador.

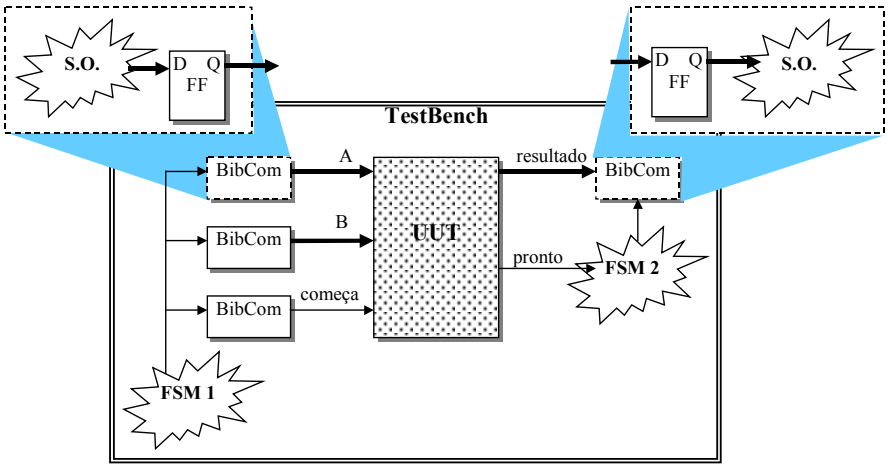

Figura 10 – Estrutura do *Testbench* do Estudo de Caso.

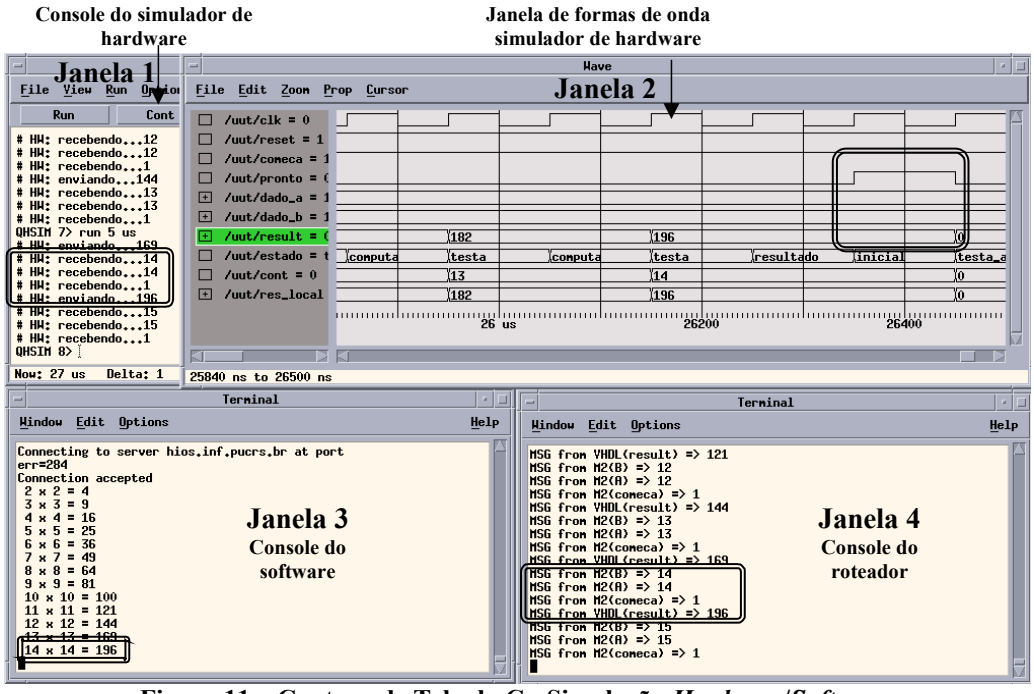

Figura 11 – Captura de Tela da Co-Simulação *Hardware*/*Software*.