Instituto de Computação
Universidade Estadual de Campinas

Metodologias de Suporte a Verificação e Análise de Modelos de Plataformas em Alto Nível de Abstração

Bruno de Carvalho Albertini

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Bruno de Carvalho Albertini e aprovada pela Banca Examinadora.

Campinas, 6 de Outubro de 2011.

Prof. Dr. Sandro Rigo (Orientador)

Prof. Dr. Guido Araújo (Co-orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR ANA REGINA MACHADO – CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA – UNICAMP

AL14m	Albertini, Bruno de Carvalho, 1980- Metodologias de suporte a verificação e análise de modelos de plataformas em alto nível de abstração / Bruno de Carvalho Albertini. – Campinas, SP : [s.n.], 2011. Orientador: Sandro Rigo. Coorientador: Guido Costa Souza de Araújo. Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação. 1. Depuração na computação. 2. Reflexão (Computação). 3. Arquitetura de computador. 4. Hardware - Engenharia de sistemas. I. Rigo, Sandro, 1976-. II. Araújo, Guido Costa Souza de, 1962-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

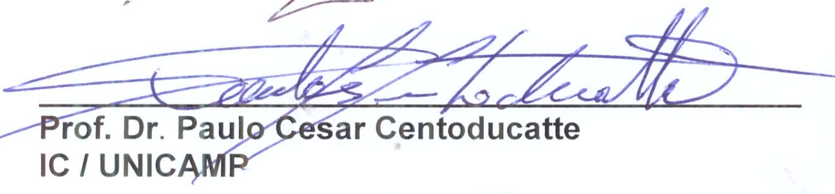
Título em inglês: Analysis and verification support methodologies for high abstractions level platforms
Palavras-chave em inglês:
Debugging in computer science
Reflection (Computer science)
Computer architecture
Hardware - Systems engineering
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Sandro Rigo [Orientador]
Rodolfo Jardim de Azevedo
Paulo Cesar Centoducatte
Luiz Cláudio Villar dos Santos
Ney Laert Vilar Calazans
Data da defesa: 06-10-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 06 de outubro de 2011; pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Luiz Cláudio Villar dos Santos
INE / UFSC

Prof. Dr. Ney Laert Vilar Calazans
Faculdade de Informática / PUC-RS



Prof. Dr. Paulo Cesar Centoducatte
IC / UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Prof. Dr. Sandro Rigo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Metodologias de Suporte a Verificação e Análise de Modelos de Plataformas em Alto Nível de Abstração

Bruno de Carvalho Albertini[1]

Outubro de 2011

Banca Examinadora:

- Prof. Dr. Sandro Rigo (Orientador)
- Prof. Dr. Rodolfo Jardim de Azevedo
  Unicamp
- Prof. Dr. Paulo Cesar Centoducatte
  Unicamp
- Prof. Dr. Luiz Cláudio V. dos Santos
  Universidade Federal de Santa Catarina
- Prof. Dr. Ney Laert V. Calazans
  Pontifícia Universidade Católica do Rio Grande do Sul
- Prof. Dr. Edson Borin
  Unicamp (Suplente)
- Prof. Dr. Mario Lúcio Côrtes
  Unicamp (Suplente)
- Prof. Dr. Roberto A. Hexsel
  Universidade Federal do Paraná (Suplente)

[1]Suporte financeiro da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP, processo 2007/58129), CNPq (processo 141774/2007-0) e suporte material do Laboratório de Sistemas Computacionais (LSC) da Universidade Estadual de Campinas (UNICAMP)

Resumo

A crescente complexidade das descrições de hardware em alto nível tem motivado a criação de metodologias de desenvolvimento por vários anos, sendo o mais recente nível de abstração representado pelo que é chamado de projeto *Electronic System Level* (ESL) e os projetos baseados em plataformas. Neste cenário, a exploração simultânea de diversos modelos arquiteturais, como os *Systems-on-Chip* (SoC), é a chave para se obter um bom balanceamento no particionamento *hardware-software* e melhorar o desempenho tanto do *hardware* quanto do *software*. Isto demanda uma infraestrutura de simulação de plataformas capaz de simular com rapidez, em um alto nível de abstração, tanto o software quanto o hardware. SystemC despontou como uma das linguagens de descrição mais adotadas e, juntamente com a modelagem em nível de transação (TLM, do inglês *Transaction Level Modeling*), vem sendo amplamente reconhecido como a técnica mais propícia para desenvolvimento em ESL. Uma das características mais marcantes de TLM é a possibilidade de reutilizar toda a infraestrutura da plataforma para a simulação de hardware e software [12].

A integração da verificação no fluxo de projeto é muito importante em uma metodologia baseada em TLM. Uma das técnicas de verificação mais conhecidas é a injeção de estímulos, usada para guiar a simulação para casos limite. Este tipo de funcionalidade é útil para aumentar a cobertura da verificação. As ferramentas disponíveis para descrições SystemC não permitem injeção de estímulos sem que o modelo seja alterado ou sem a utilização de um núcleo de simulação modificado para tal tarefa. Para a depuração, não temos notícia de nenhuma ferramenta de código aberto disponível, porém existem boas ferramentas comerciais preparadas especificamente para a depuração de modelos em SystemC.

Nesta tese são propostas três metodologias para melhorar a capacidade de introspecção, depuração e análise de modelos de *hardware* descritos em alto nível de abstração. A primeira delas é composta por uma metodologia de reflexão computacional aplicável a módulos SystemC através da inserção de módulos de inspeção, que chamamos de *ReflexBox*. A segunda técnica desenvolvida foi chamada de *SignalReplay*, e representa uma evolução da primeira técnica voltada para a captura, análise e injeção dos dados coletados

pela reflexão. A última metodologia proposta, chamada de PDFA (do inglês, *Platform Dataflow Analysis*) visa extrair metadados através da reflexão de tipos sobrecarregados, permitindo que o projetista aplique técnicas de compiladores para a análise de *hardware*.

Os resultados obtidos são apresentados como experimentos, implementados na forma de estudos de caso. Estes experimentos permitiram avaliar a eficácia das técnicas propostas que, ao contrário de trabalhos correlatos, aderem a seis princípios que consideramos fundamentais: (1) não são intrusivas em relação às modificações no modelo que podem ser necessárias para implementar a introspecção; (2) não necessitam de modificações no ambiente de simulação, compiladores ou bibliotecas, incluindo nossa linguagem alvo: SystemC; (3) geram uma sobrecarga pequena no tempo de simulação; (4) proveem observabilidade e controlabilidade; (5) são extensíveis, permitindo a adaptação para utilização em trabalhos similares, com pouca ou nenhuma modificação nas metodologias; e (6) protegem a propriedade intelectual do módulo sob verificação.

Abstract

The increasing complexity of high level hardware descriptions has motivated the creation of development methodologies for several years, being the most recent level of abstraction represented by projects based on platforms and on the so called Electronic System Level design (ESL). In this scenario, simultaneously exploring different architectural models, like Systems-on-Chip (SoC), is the key to achieve a good balance on hardware-software partitioning and improve performance of both hardware and software. This requires a platform simulation infrastructure able to simulate at high speeds and high level of abstraction, both software and hardware. SystemC emerged as one of the most widely adopted description languages and, when used with the Transaction Level Modeling (TLM), has been widely recognized as the most suitable for ESL development. One of the most striking features of TLM is the possibility to reuse all the infrastructure platform for the simulation of hardware and software [12].

Integration of the verification into design flow is a key point in a TLM-based methodology. One well-known verification technique is the injection stimuli, used to guide the simulation to borderline states. This kind of functionality is useful to increase the coverage of the verification. The tools currently available for SystemC descriptions do not allow stimuli injection without model modifications, or without the use of a modified SystemC simulation core specially crafted for this task. We could not find any open source tool for debugging, but there are good commercial tools specifically prepared to SystemC model debugging.

This thesis proposes three methodologies focused on improving the support for introspection, debug, and analysis of hardware models described in high abstraction level. First one is a methodology using computational reflection, applicable to SystemC descriptions by inserting inspection modules, that we call *ReflexBoxes*. The second technique is called *SignalReplay*, an evolution of the first technique focused on the capture, injection, and analysis of data collected by reflection. The last proposed methodology, called Platform Dataflow Analysis (PDFA), aims on the metadata extraction through overloaded type reflection, allowing the designer to use compiler techniques for hardware analysis.

The results are presented as experiments, implemented as case studies. These expe-

riments allowed us to evaluate the effectiveness of the proposed techniques that, unlike related work, adhere to what we consider six fundamental principles: (1) are not intrusive regarding any model modifications that may be necessary to implement introspection; (2) do not require any change in simulation environment, compilers, or libraries, including our target language: SystemC; (3) generate minimal overhead in simulation time; (4) provide observability and controllability; (5) are extensible, allowing the adaptation for use in similar work with little or no change in the methodology; and (6) protect the intellectual property of the module under verification.

Agradecimentos

Agradeço ao CNPq e à FAPESP, por proporcionar o suporte financeiro que me manteve no doutorado durante os anos iniciais. Também à Unicamp, em especial ao LSC (Laboratório de Sistemas Computacionais) do Instituto de Computação, do qual utilizei intensivamente os recursos para a implementação das ideias aqui expostas.

Aos professores, Côrtes, Pannain, Ducatte e Rodolfo, que me ensinaram a maior parte do que sei hoje sobre a minha área. Um agradecimento especial ao Guido e ao Sandro, que são os responsáveis pelo amadurecimento das ideias, pelas horas de discussão e paciência em me orientar.

À minha família, que me proveu o suporte necessário, aos meus pais que me deram os genes e a minha namorada, que graciosamente cedeu seu tempo comigo em prol do doutorado em diversas ocasiões.

Por último, aos colegas de laboratório pelo convívio diário, pelas discussões de ideias e pelo companheirismo, dentro e fora do ambiente acadêmico. Até mais e obrigado pelos peixes.

Sumário

Resumo **v**

Abstract **vii**

Agradecimentos **ix**

1 Introdução **1**

2 Trabalhos Relacionados **9**

3 Introspecção em Plataformas SystemC **15**

3.1 Depuração e Verificação de Modelos em SystemC 15
3.2 Por que a Reflexão Computacional? 16
3.3 Introspecção em Módulos SystemC 18
3.4 Depurando uma Plataforma Exemplo Através de Simulação 21
3.5 Verificação de Módulos Isolados 30
3.5.1 *Signal Replay* 31
3.6 Conclusão 37

4 Análise de Fluxo de Dados em Plataformas **39**

4.1 Análise de Fluxo de Dados 39
4.1.1 Conjuntos *generate* e *kill* 41
4.1.2 Conjuntos *input* e *output* 43
4.2 Análise de Fluxo de Dados em Plataformas 44
4.2.1 Conjuntos *generate* e *kill* 45
4.2.2 Conjuntos *input* e *output* 46
4.2.3 Comentários Sobre a Implementação 46
4.2.4 Interações Entre os IPs 47
4.3 Estudo de Caso #1 48
4.4 Estudo de Caso #2 51

4.5 Experimentos 55
4.6 Conclusão 56

5 Conclusão e Trabalhos Futuros **58**
5.1 Trabalhos Futuros 59
5.2 Lista de Publicações 60

A *Links* **62**

Bibliografia **65**

Lista de Tabelas

3.1 Tempo médio de simulação (segundos) do MP3 com *Signal Replay*, desvio padrão (DP) e ganho 35

4.1 O conjuntos *generate* e *kill* para o código de exemplo 42
4.2 O conjuntos *generate* e *kill* para os blocos básicos do exemplo 43
4.3 O conjuntos *in* e *out* para os blocos básicos do exemplo 44
4.4 O conjuntos *generate* e *kill* para o estudo de caso #1 51
4.5 Conjuntos *in* e *out* para o estudo de caso #2 53
4.6 Tempos de execução para o estudo de caso #2 55

Lista de Figuras

1.1 Níveis de abstração usados na modelagem de *hardware* 2
1.2 Fluxo básico de desenvolvimento ESL 6

3.1 Fluxo de reflexão usando a biblioteca Reflex-SEAL 19
3.2 Arquitetura da Plataforma MP3 `dist10` modificada 23
3.3 Trecho do dicionário de dados para o IP `ac_tlm_imdct` 24
3.4 O arquivo `main.cpp` representando a configuração da plataforma MP3 usada no exemplo 25
3.5 Interagindo com o *ReflexBox* através de `telnet` 26
3.6 Exemplo de execução do *ReflexBox* 28
3.7 Inserindo um `breakpoint` no *ReflexBox* 28
3.8 Mudando a condição de parada do *ReflexBox* 29
3.9 Atributos refletíveis do módulo IMDCT, mostrados pelo *ReflexBox* 29
3.10 Saída da introspecção do módulo IMDCT 29
3.11 Reprodução da simulação do estudo de caso para o *Signal Replay* 32
3.12 Exemplo de Plataforma DUV para *scoreboard* 33
3.13 A plataforma de estudo para o *Signal Replay* instrumentada usando reflexão 35

4.1 Trecho de código com as sentenças numeradas de s_1 a s_{10}. 40
4.2 Definições contidas no trecho de código, numeradas de d_1 a d_7. 40
4.3 CFG para o trecho de código da Figura 4.2 41
4.4 Encapsulamento to *tag* 47
4.5 CFG do estudo de caso #1 48
4.6 Plataforma do estudo de caso #1 50
4.7 Organização interna da primeira versão do IP SHA 52
4.8 A plataforma SHA 54
4.9 Tempos de execução em segundos para o estudo de caso #2 56

Capítulo 1

Introdução

Os projetos eletrônicos modernos estão se tornando cada vez mais complexos. O crescimento do mercado de dispositivos eletrônicos a partir da segunda metade da década de 90, impulsionado principalmente pela crescente utilização de dispositivos móveis, mudou a forma como a indústria eletrônica executa os projetos. Antes da revolução SoC (do inglês, *System-On-Chip*) as indústrias eletrônicas possuíam o domínio de todo o ciclo de projeto, desde a definição até a fabricação. Nos dias atuais, a identificação de uma nova oportunidade de mercado, a definição e especificação do projeto, o desenvolvimento e montagem dos diversos componentes e a fabricação do produto final são etapas cada vez mais realizadas por diferentes organizações. Chamamos esta característica de **ortogonalização** da indústria eletrônica, pois esta mudou de uma posição de projetos vertical (interna) para horizontal.

Além de focar na competência principal de cada organização envolvida, esta forma de projeto permite que as empresas trabalhem em paralelo, de forma a reduzir o tempo entre a concepção e a entrega de um produto, também chamado de TTM (do inglês, ***time-to-market***). Esta pressão do mercado pela diminuição do TTM e a complexidade crescente dos projetos, forçou os projetistas a adotarem técnicas que aumentem a produtividade, porém sem prejudicar a qualidade dos produtos. As propostas que sobreviveram até os tempos atuais possuem duas características em comum: o alto nível de abstração e a reutilização de componentes.

Os níveis de abstração dos projetos de *hardware* possuem várias nomenclaturas e diferentes classificações, começando no silício, o nível **físico**, passando pelos níveis de **circuitos**, ainda amarrado à disposição física dos componentes mas já com certa modularização, e o de **portas lógicas**. A maioria das linguagens de descrição de *hardware* permitem que um modelo seja escrito no nível de portas lógicas, porém este nível normalmente é fruto da compilação de níveis com abstração mais alta. Na prática, estes três níveis somente são utilizados quando o projetista precisa de uma caracterização precisa (ex. consumo,

atraso, tamanho, etc), normalmente na fase final de um projeto.

Quanto mais baixo o nível de abstração, mais próximo se está de um modelo sintetizável, em detrimento da velocidade com que se desenvolve o projeto, incluindo-se a verificação e depuração do mesmo. Hoje em dia, podemos dizer que os níveis físico e de circuito são usados somente em módulos isolados que possuem pré-requisitos específicos quanto a disposição física dos módulos no circuito final. Até mesmo o próximo nível, de portas lógicas, está se tornando obsoleto, principalmente devido à qualidade dos algoritmos de síntese atuais, capazes de gerar descrições sintetizáveis comparáveis às feitas pelo ser humano a partir de descrições de alto nível [6]. Dificilmente usa-se estes níveis para uma plataforma inteira pois possuem um desenvolvimento custoso, principalmente em relação ao tempo de projeto, além de serem muito propícios a erros.

Após as descrições de circuitos, os projetistas partiram para descrições de portas lógicas, suportadas por linguagens como Verilog e VHDL. Estas linguagens também abrangem os níveis de abstração de transferência entre registradores (**RTL**, do inglês *Register Transfer Level*). Linguagens mais recentes, como SystemC e SystemVerilog, suportam também o nível **funcional**, atualmente o nível de abstração mais próximo possível de uma implementação em *software* puro. Como dissemos anteriormente, o nível de portas lógicas está sendo gradualmente abandonado, sendo priorizados os níveis RTL e funcional. No nível funcional, algumas primitivas de *hardware*, como a temporização, normalmente são ignoradas de forma a se obter uma descrição comportamental do *hardware*. Com este aumento do nível de abstração, ganha-se a velocidade de simulação e a facilidade de depuração, em detrimento da síntese e precisão da caracterização.

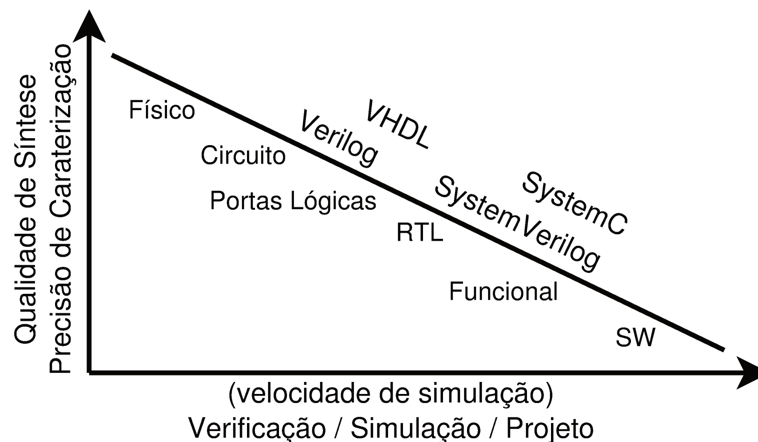


Figura 1.1: Níveis de abstração usados na modelagem de *hardware*

Na Figura 1.1, mostramos uma visão geral dos níveis de abstração e as quatro linguagens mais utilizadas para descrições de *hardware*. Note que existe uma troca entre a qualidade da síntese e a velocidade de simulação. A primeira reflete diretamente na

precisão da caracterização e a segunda nas facilidades proporcionadas pelo maior nível de abstração, tais como maior velocidade de simulação e menor tempo de projeto (incluindo o tempo necessário para se efetuar mudanças).

Mesmo com o nível de abstração das descrições de *hardware* próximo a um *software*, a busca pela diminuição do TTM continha pressionando a indústria. O **reuso de componentes**, em todos os níveis de abstração, foi a solução encontrada para aumentar a confiabilidade do produto final e diminuir ainda mais o tempo de projeto. A reutilização de componentes sempre existiu, porém até a popularização das descrições em RTL, era utilizada com restrições pois, nos níveis de abstração mais baixos, os módulos possuem características que dificultam a portabilidade dos mesmos (ex. a interface de acesso externo). Com a adoção de descrições RTL, os projetistas passaram a contar com um grau maior de portabilidade, o que aumentou a reutilização devido a facilidade com que este nível de abstração proporciona na integração e personalização dos módulos. Com a portabilidade facilitada, os projetistas adotaram uma modificação do reuso, herdada do desenvolvimento de *software*: a utilização de componentes pré-validados. Estes módulos são chamados de IPs (do inglês, *Intelectual Property*), e normalmente são fornecidos em diferentes níveis de abstração, equivalentes funcionalmente entre si. Estes componentes são escritos de forma a possuírem características específicas de desempenho e consumo de energia, validados, e então fornecidos na forma de um módulo que o projetista pode integrar em sua plataforma. Nos tempos atuais, é possível comprar um IP com uma (ou mais) descrição sintetizável e sua equivalente funcional, usada para fins de simulação. O desenvolvedor do IP garante o funcionamento do módulo através de testes e feitos com este propósito. Normalmente, os vetores de testes utilizados são disponibilizados junto com o módulo, com informações sobre as simulações realizadas para validação, possibilitando que o desenvolvedor assuma que o módulo atende os requisitos necessários.

A reutilização de IPs possibilitou que o desenvolvedor construa plataformas baseadas total ou parcialmente em componentes prontos, chamados de componentes de prateleira (do inglês, *off-the-shelf*). Chamamos esta abordagem de desenvolvimento baseado em plataformas (do inglês, *Platform Based Design*). Uma plataforma é um conjunto de recursos de *hardware* e *software* capazes de operar conjuntamente para prover funcionalidade a uma classe de aplicações. Nesta forma de projeto, o projetista divide o sistema final em módulos funcionais, onde cada módulo realiza uma tarefa que contribui para o resultado final. Com a plataforma dividida em módulos, o projetista pode então experimentar diversas configurações da mesma, de forma a atingir os requisitos do produto final. Dois exemplos de exploração de plataformas são: a infraestrutura de comunicação entre os módulos e o particionamento *hardware-software*.

Particionamento é como se conhece a decisão de tornar um módulo um *hardware* ou executar a tarefa equivalente em *software*. Devido a abundância de processadores e a

facilidade de utilização (disponibilidade de compiladores, ferramentas de depuração e integração), a decisão de transformar um módulo em *hardware* está corriqueiramente ligada a requisitos de latência e consumo. Em relação a reutilização de módulos, é desejável que se utilize o máximo possível de módulos prontos e desenvolva-se somente módulos específicos. Apesar de a reutilização de IPs pré-validados ajudar a reduzir o trabalho de verificação dos módulos isoladamente, gera uma sobrecarga na validação da plataforma como um todo pois a combinação de vários IPs complexos pode levar a erros não previsíveis e difíceis de serem detectados.

Para descrever plataformas, criaram-se níveis de abstração baseados nos existentes para a descrição de *hardware*. O mais conhecido deles é o TLM (do inglês, *Transaction Level Modeling*). TLM utiliza o mesmo nível de abstração do nível funcional, mas também abstrai a comunicação entre os módulos, tornando-a chamadas de função. Como afirma Ghenassia no seu livro sobre técnicas TLM [12], um dos aspectos mais atraentes do nível TLM é a possibilidade de reutilizar toda a infraestrutura de uma plataforma para o desenvolvimento e verificação tanto do *software* quanto do *hardware*. Isso significa que a mesma plataforma em alto nível é utilizada como modelo de referência no fluxo de verificação funcional para o desenvolvimento do *hardware*, e também pelos desenvolvedores de *software* como uma especificação executável do sistema. Além disso, outro fator importante para uma descrição de *hardware* proporcionado pelo TLM, é a integração precoce da verificação no ciclo de desenvolvimento.

No âmbito de descrições de plataformas, chamamos de ESL (do inglês *Electronic System Level*) o conjunto de abstrações, linguagens e metodologias capazes de descrever plataformas em um alto nível de abstração (mesmo que utilizando vários níveis em uma mesma plataforma). No livro [27], capítulo 1, seção 1.2, os autores detalham os requisitos para que uma descrição possa ser chamada de ESL. Usando ESL, o projetista de *hardware* pode construir sua plataforma modularmente, escrevendo seus próprios módulos, infraestrutura de conexão e processadores. O objetivo principal é a simulação do projeto antes que o mesmo seja sintetizado, possibilitando a verificação, depuração e caracterização em alto nível, diminuindo assim os custos do projeto e o tempo de desenvolvimento.

Bailey e outros [3] dividem a capacidade de depuração e análise em três grupos distintos. A observabilidade é simplesmente a habilidade de identificar o estado interno e valores instantâneos de grandezas do sistema. A controlabilidade significa prover ao projetista os meios necessários para que este possa conduzir o sistema a atingir um estado interno preciso, de forma a aumentar o controle sobre o mesmo. A corretude é a capacidade de ajudar o projetista a corrigir erros descobertos, o que é mais fácil no nível ESL que no nível do silício, porém ainda não suportado completamente pelas ferramentas ESL.

Quanto mais alto o nível de abstração, mais fácil se torna conseguir boas ferramentas de depuração e análise. É sabido que em um sistema em silício, os únicos pontos que

podem ser observados são as interfaces externas e os pontos de observação, determinados ainda no projeto. No caso das descrições de *hardware*, a limitação da observabilidade e da controlabilidade está somente ligada a capacidade da linguagem e do ambiente utilizado de proverem mecanismos. Nas tecnologias ESL, os altos níveis de abstração utilizados nas descrições não devem ser somente vistos como atrativos devido à velocidade de simulação, mas também pela flexibilidade de exploração e recursos para depuração e análise (ex. substituição de IPs por equivalentes funcionais, introspecção interna dos módulos, mudança de topologia de comunicação, etc). Esta visão de uma plataforma como alvo de projeto cria uma demanda por novas funcionalidades relacionadas com a verificação em modelos de plataformas descritos em TLM. Os objetivos principais neste cenário são:

- aumentar a **cobertura da verificação** visando exercitar o maior número possível de caminhos tanto do *hardware* quanto do *software*, suficiente para que o módulo seja considerado correto;

- aumentar as possibilidades de **interação** do projetista com a plataforma, por exemplo guiando o fluxo de simulação para casos extremos de teste usando injeção de estímulos escolhidos propositalmente;

- tornar o ambiente de simulação mais amigável durante a **depuração** da plataforma, permitindo pontos de parada da simulação automáticos e introspecção da plataforma.

O ciclo de desenvolvimento mais comum de uma plataforma usando ESL é descendente (da mais alta abstração para a mais baixa), podendo ser dividido em duas fases distintas, conforme mostrado na Figura 1.2. A primeira consiste em montar uma especificação no mais alto nível possível, senão em *software*, para viabilização do projeto. Esta mesma fase gera um modelo da plataforma ou uma especificação executável da plataforma, de onde se retira a partição ideal entre *software* e *hardware* (entenda-se: decidir qual parte do modelo irá executar como *software* dentro de um processador e qual parte será executada em *hardware*). Com a abundante disponibilidade atual de processadores de uso geral e específico, esta decisão é normalmente tomada com base no tempo de entrega dos resultados do módulo (latência) e no consumo de energia do mesmo, mantendo-se em *software* os módulos que atingirem os requisitos mesmo quando executados em um processador.

A segunda fase possui dois fluxos de projeto. O primeiro diz respeito ao desenvolvimento do *software*, também chamado de *firmware* ou *midlet*, quando presente. Quase sempre, este fluxo consiste em desenvolver ou adaptar os módulos que executarão parte das funcionalidades na forma de *software* executando em um ou mais processadores, gerando o que chamamos de *software-IP*, um processador encapsulado que faz o trabalho que seria feito por um *hardware*. No outro fluxo, desenvolve-se o *hardware*, montando-se

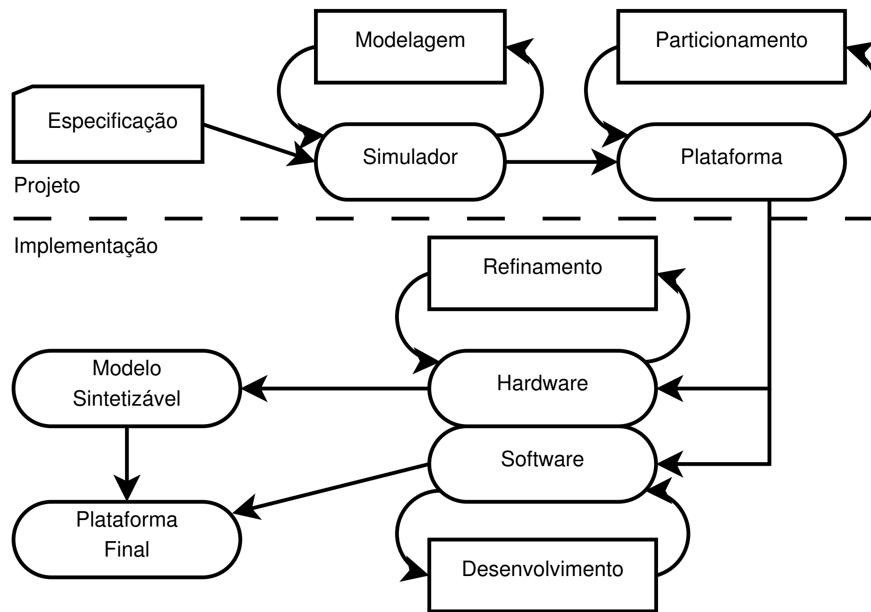


Figura 1.2: Fluxo básico de desenvolvimento ESL

a plataforma através de módulos disponíveis ou descrevendo cada módulo, conectando-os por meio de estruturas específicas para tal (ex. barramentos). Neste fluxo existe um ciclo importante que chamamos de refinamento de IP. Na primeira rodada do fluxo, o que se tem é a especificação executável em alto nível, resultado da primeira fase, que pode ser usada como base para a elaboração da plataforma. Como esta é a primeira especificação executável, dificilmente é sintetizável. A cada iteração deste ciclo, o projetista faz uma descrição equivalente do módulo em uma abstração de mais baixo nível, alterando a linguagem de descrição se necessário, até atingir o nível sintetizável ou com detalhamento suficiente para que seja feita a caracterização da plataforma.

A grande vantagem do ESL em relação ao particionamento é que os dois fluxos podem ser feitos ao mesmo tempo. A partir da primeira iteração do fluxo de *hardware*, os projetistas de *software* já possuem um modelo estável e simulável da plataforma, com uma descrição do *hardware* condizente com o que será gerado em silício. Mais que isso, o desenvolvedor de *software* pode adotar qualquer uma das descrições oriundas das iterações de refinamento, incluindo a possibilidade de misturar descrições do mesmo módulo com diferentes níveis de abstração na mesma plataforma, de modo a obter o detalhamento do *hardware* que precisa para o desenvolvimento do *software*. Se o projetista está testando determinada funcionalidade, pode necessitar de um detalhamento maior no módulo alvo, porém caso não tenha interesse, pode usar uma versão funcional deste, de forma a obter um tempo de simulação mais rápido. O desafio dos desenvolvedores de ferramentas voltadas para ESL, é justamente manter a equivalência e a proximidade do *hardware* real mesmo

nos níveis de abstração mais altos.

A simulação virtual de plataformas [20], a verificação por asserções [17, 9, 15] e a introspecção [11, 8] são técnicas que têm sido adotadas como suporte para a exploração de plataformas escritas em linguagens de descrição de *hardware*, como SystemC. Notadamente, o que se busca com estas técnicas é analisar, caracterizar e validar (incluindo teste, depuração e verificação) as representações de projeto ainda no domínio do *software*.

Um dos problemas que encontramos é que estas técnicas usualmente são específicas de cada fabricante de ferramentas ESL ou específicas para uma determinada plataforma, não permitindo uma análise extensiva do comportamento da plataforma através do *software*, *hardware* e a comunicação entre estes dois domínios. Idealmente o projetista deveria ser capaz de rastrear qualquer valor (ex. contido em um registrador ou em uma variável) até sua origem, possivelmente em outro módulo, seja este um módulo de *hardware* ou *software*, independentemente da quantidade de outros módulos que estejam no caminho. Esta tarefa pode ser realizada por técnicas baseadas no rastreamento de dados [18], porém a análise resultante deste tipo de técnica não fornece informações suficientes para uma ampla verificação ou otimização da plataforma.

O objetivo principal desta tese é propor uma solução para alguns dos problemas encontrados na exploração de plataformas de descrição de *hardware* em alto nível. As contribuições deste trabalho serão apresentadas divididas em três fases. A primeira fase está relacionada à uma base para exploração de plataformas, consistindo em um arcabouço de ferramentas de apoio às técnicas propostas. Desenvolvemos e aplicamos uma metodologia de exploração ESL baseada na reflexão computacional como meio de obtenção das informações necessárias e injeção de sinais. Sobre esta metodologia e como complemento da mesma, desenvolvemos um módulo em SystemC capaz de utilizar a metodologia proposta e aplicá-la a qualquer módulo SystemC, em qualquer nível de abstração, sem se tornar intrusivo e com pouca interação do usuário. Esta metodologia, bem como o módulo de introspecção, estão descritos no Capítulo 3.

A segunda fase corresponde a uma adaptação da metodologia de reflexão, aplicada especificamente no modelo de verificação chamado de DUV. Na metodologia proposta, mostraremos como se utilizar as técnicas de reflexão desenvolvidas para melhorar a velocidade de simulação quando utilizada para verificação de módulos isolados da plataforma. Os vetores de teste são retirados da própria plataforma, através de uma simulação confiável, normalmente feita com o modelo de referência, com a especificação executável do sistema ou com uma plataforma onde o módulo alvo esteja descrito em um nível de abstração maior (no caso do refinamento). Chamamos esta metodologia de *Signal Replay* e pode-se encontrá-la na seção Seção 3.5.

A terceira e última fase corresponde a uma proposta para a análise de fluxo de dados em plataformas, usando a metodologia proposta no Capítulo 3 aliada à sobrecarga de

tipos para implementarmos uma adaptação da teoria de análise de fluxo de dados (compiladores). No Capítulo 4 mostramos a metodologia e dois exemplos práticos de aplicação voltados para a depuração de plataformas e para a análise de paralelismo em plataformas.

Capítulo 2

Trabalhos Relacionados

Ferramentas Comerciais [1]

A ARM tem entre seus produtos uma ferramenta chamada *RealView Development* que, entre outras coisas, possibilita a depuração de modelos em SystemC em tempo real, incluindo exploração de plataformas TLM, depuração remota, onde é possível depurar uma simulação executando em uma máquina diferente da aplicação, e de baixo nível (esta última somente para modelos de processadores ARM).

O programa *Vista Architecture* é uma das ferramentas do pacote *Vista*, da *Mentor Graphics*. Este pacote de ferramentas possui o depurador para projetos em SystemC com o maior número de funcionalidades dentre as ferramentas avaliadas, permitindo que o projetista faça o *trace* das transações, veja a sequência de eventos e acompanhe a execução dos processos SystemC passo a passo. Através da própria ferramenta pode-se analisar o desempenho de um modelo específico na plataforma, considerando a implementação em *hardware* e na forma de um IP de *software* (é possível analisar o desempenho tanto no domínio do tempo quanto da potência).

O *Platform Architect* da *Synopsys* (originalmente da *CoWare*), possui a mesma abordagem que as ferramentas da Mentor, com foco na exploração de plataformas em alto nível e monitores de dados. A empresa possui um ambiente gráfico específico para a modularização de projetos, usando modelos C/C++ como ponto de partida de uma especificação executável. Hoje, esta ferramenta destaca-se pela sua capacidade de visualização gráfica de pacotes TLM, o que beneficia o projeto de arquiteturas de interconexão complexas, como NoCs.

A *Cadence* possui um conjunto de ferramentas chamadas *Incisive Palladium* que possui várias ferramentas de introspecção. A versão do SystemC fornecida com as ferramentas

[1] **Consulte o Apêndice A para referências às ferramentas citadas.**

suporta asserções dinâmicas, uma versão da empresa para asserções condicionais, porém com parâmetros alteráveis em tempo de simulação; instrumentação de código, para inserção de pontos de depuração, *breakpoints*, etc; e depuração em tempo de execução. Há inclusive uma ferramenta chamada *Incisive Simulator* que, entre outras funcionalidades, é capaz de instrumentar o código automaticamente para permitir a geração automática de casos de teste.

Uma empresa japonesa chamada *CATS*, possui uma ferramenta de depuração para SystemC chamada *XModelink SystemC Debugger*. A ferramenta possui as operações básicas de depuração como *watches*, *traces*, etc., similares aos seus concorrentes, porém com limitações em relação às opções de depuração (ex. nem todos os tipos de dados podem ser inspecionados).

O problema principal das ferramentas comerciais é que muitas vezes elas utilizam versões proprietárias da biblioteca SystemC e da biblioteca de comunicação TLM, seja total ou parcialmente. Estas versões são especialmente projetadas para ajudar na depuração por meio de instrumentação das bibliotecas. A utilização destas versões instrumentadas está fortemente ligada à ferramenta oferecida. Na maior parte das vezes, o núcleo do SystemC adere à padronização da OSCI, porém com primitivas extras sobre o padrão, fazendo com que projetos específicos sejam amarrados à ferramenta e a portabilidade de IPs seja degradada. Devemos ainda considerar que a integração com qualquer código legado é difícil a ponto de inviabilizar a integração frente a um projeto completamente novo.

Também não há entre as ferramentas comerciais uma padronização dos testes, portanto não é possível utilizar um caso de teste gerado em uma ferramenta para alimentar um IP executando em outra sem antes adaptar os arquivos de dados para teste (*traces*).

Na medida do possível, avaliamos as ferramentas comerciais disponíveis, através dos convênios do Laboratório de Sistemas Computacionais (IC-UNICAMP) ou de versões disponíveis para avaliação. Nosso propósito desde o início do projeto foi ater-se estritamente às especificações padronizadas pela OSCI e usar somente ferramentas disponíveis gratuitamente, preferencialmente de código aberto. Por este motivo e pelo SystemC estendido ou fora de padrão, descartamos neste projeto a utilização das ferramentas comerciais avaliadas. Cabe ressaltar que, apesar de todas as ferramentas avaliadas oferecerem algum tipo de introspecção, nenhum mecanismo comercial é similar ao proposto neste trabalho.

Trabalhos Acadêmicos

Rogin e outros [28] apresentaram um ambiente voltado para a depuração SystemC. Seu trabalho é baseado em uma biblioteca que sobrescreve o núcleo de simulação original do SystemC para capturar as informações de depuração. Além disso, este trabalho introduz

uma representação intermediária dos módulos de *hardware* desenvolvidos, permitindo um mapeamento da simulação com a descrição do modelo, em tempo de execução. A representação intermediária desenvolvida é compatível com o GDB, possibilitando que sejam desenvolvidas ferramentas gráficas para auxiliar no trabalho de depuração. De fato, os autores apresentam um exemplo de tal ferramenta. É importante salientar que a metodologia apresentada somente é aplicável a modelos em SystemC cujo código-fonte seja integralmente conhecido e disponível em tempo de execução, o que nem sempre é verdade quando se trabalha com bibliotecas comerciais, cujo código fonte é protegido.

SystemCXML [23] e *LusSy* [26] são ferramentas de código aberto voltadas para a visualização da hierarquia de conexão dos módulos SystemC. Apesar de proverem extração de metadados para módulos SystemC, não é possível depurar em tempo de execução sem que o projetista escreva os próprios módulos de depuração. Para esta abordagem de depuração em tempo de execução, pudemos encontrar somente o GDB e ferramentas derivadas do mesmo. O problema do GDB é que este se torna muito difícil de utilizar em plataformas, principalmente porquê este programa foi feito para depuração de um código genérico, não distinguindo assim o código da biblioteca SystemC do código do usuário. Os depuradores trazidos do domínio de *software* para o ambiente de prototipação de *hardware*, normalmente consideram todo o código de uma especificação executável, mesmo escrita em SystemC, como um único fluxo de execução. Se o projetista não tomar o devido cuidado, pode acabar com o depurador inspecionando o código interno da biblioteca, mostrando assim o código das classes de SystemC e não o código que o projetista quer realmente depurar. Este fato, somado à complexidade da biblioteca SystemC, desencoraja a utilização de depuradores convencionais para C/C++, mesmo em projetos simples.

Até o momento, os trabalhos com a SCV (*SystemC Verification Library*) limitam-se à verificação de modelos RTL. Sabemos pelas discussões nos grupos de interesse que há a intenção de também fornecer suporte para a depuração em alto nível em versões futuras, porém os trabalhos ainda não estão em uma fase madura o suficiente para serem utilizáveis, restringindo-se, no nível funcional, a verificações declarativas do tipo *assert*. Também nota-se que a SCV não está seguindo os planos de projeto da OSCI. A versão atual não é compatível diretamente com os compiladores atuais e não há previsão de atualização da biblioteca.

Lapalme e outros [19] desenvolveram um ambiente de prototipação de *hardware* baseado em uma linguagem de descrição chamada de ESys.NET [24], uma extensão da linguagem .NET (baseada em C#) para descrições de *hardware*. Eles foram pioneiros em identificar a necessidade de introspecção de código para propósitos de depuração e verificação mas, para eles, a introspecção é facilmente obtida pois C# possui suporte nativo à reflexão. Apesar de ter os trabalhos reconhecidos em publicações da área (ex. *Transactions on Embedded Computing Systems*, ACM), o grupo baseou-se na linguagem

C# como base de prototipação, ignorando as linguagens emergentes como SystemC e SystemVerilog. Acreditamos que a baixa aceitação do ambiente se deve ao fato de o mesmo ser fortemente ligado a um único sistema operacional e representar uma grande mudança de paradigma em relação às linguagens usadas nos níveis de abstração anteriores. Com o surgimento de SystemC, a utilização da ESys.NET está restrita a um pequeno grupo de pesquisadores, gerando dúvidas em relação a sua continuidade.

Déharbe e Medeiros [7] mostraram outra forma de se obter introspecção usando programação orientada a aspectos. A ideia principal do artigo é a instrumentação de todo o código para gerar métricas de codificação, porém a mesma técnica pode ser aplicada para propósitos de verificação e depuração. A instrumentação foi realizada usando a biblioteca *AspectC++*, uma extensão de C++ largamente utilizada como meio de obtenção deste paradigma de programação para esta linguagem de programação. A principal desvantagem desta abordagem é que os projetistas precisam ter conhecimento que o módulo a ser projetado será acessado usando esta metodologia e escrever todo o código usando o paradigma de orientação à aspectos (AOP, do inglês *Aspect Oriented Paradigm*) que, na maior parte dos casos, significa que os projetistas precisam aprender um novo paradigma de programação. Outro problema da técnica é que o código legado deve ser adaptado de forma a adotar os mesmos princípios, tornando difícil a adoção em massa da metodologia proposta para o domínio da modelagem de *hardware*.

O ReSP [4] é uma proposta de uma ferramenta para depuração de modelos de SystemC que utiliza a linguagem Python para a obtenção da reflexão. O núcleo de simulação do SystemC e os modelos do usuário são mantidos sem modificações. O ambiente gera automaticamente um código envoltório sobre cada módulo SystemC (do inglês, *wrapper*) usando uma biblioteca de conexão de código C/C++ com a linguagem Python. Como esta última possui reflexão nativa, seus autores beneficiaram-se desta para implementar a introspecção. Em tese, a principal desvantagem deste trabalho é a sobrecarga na velocidade de simulação imposta pelo adaptador entre o ambiente de execução C/C++ (onde se executam os módulos em SystemC) e o ambiente interpretado do Python. Não obtivemos dados concretos sobre a sobrecarga deste projeto por meio de simulações próprias, pois o ambiente não é público. No artigo (e em outros relacionados) os autores afirmam que esta sobrecarga é de cerca de 1% do tempo total de simulação, porém o adaptador que os autores utilizaram gera uma sobrecarga dependente do número de chamadas entre os dois ambientes de execução. Para a característica de introspecção, são realizadas diversas chamadas de leitura e escrita em cada ciclo de simulação, motivo pelo qual acreditamos que esta técnica somente se aplique à introspecção de pequenos conjuntos de dados que não necessitam ser monitorados frequentemente. Este é o trabalho que hoje mais se assemelha ao proposto no Capítulo 3 desta tese.

Mathaikutty [23] e Shukla propõem uma maneira de classificar IPs de acordo com

metadados extraídos diretamente do código fonte dos módulos SystemC. A técnica para extrair os metadados utiliza reflexão computacional através de uma ferramenta chamada KaSCPar, que extrai os metadados do IP SystemC para um formato XML. O principal objetivo é a composição de IPs usando uma biblioteca de IPs previamente analisados, porém a abordagem com a reflexão é similar a que adotamos para nossos propósitos.

Misera [25] e outros mostraram que as metodologias de injeção de falhas podem ser aplicadas em SystemC para verificação. No artigo, os autores aplicam diversas técnicas de engenharia de *software* para injeção de falhas, como a inserção de mutantes e sabotadores. A injeção é feita explitamente, alterando o código para que o módulo aceite estímulos de verificação. A técnica que mostraremos possui a vantagem de injetar estímulos sem modificação no módulo. Não aplicamos as técnicas de verificação propostas, porém a metodologia que apresentaremos pode ser usada para os mesmos propósitos. Já Lisherness e Chang [21] apresentam SCEMIT, uma ferramenta para gerar e injetar automaticamente erros em plataformas SystemC. O foco é a técnica de injeção de mutantes para garantir a cobertura de testes gerados automaticamente. A injeção de mutantes consiste na inserção de códigos ou valores que simulem um estado específico do sistema, desejado para fins de verificação (ex.: forçar a tomada de um caminho condicional forçando um valor específico na variável de controle). Destacamos neste artigo a técnica utilizada para a injeção de estímulos: os autores criaram um módulo para um compilador capaz de injetar automaticamente os estímulos gerados. Esta técnica se mostrou mais rápida que as demais como asserções dinâmicas, modificações no núcleo de simulação e a reflexão, incluindo a que desenvolvemos. A técnica porém, possui seu uso limitado à injeção de sinais, não sendo genérica o suficiente para outras utilizações, mesmo relativas à depuração e à verificação. Além disso, é necessária a alteração do compilador, o que viola um dos nossos objetivos.

Chugh e outros [5] mostram uma forma de detectar e analisar condições de disputa de dados em programas concorrentes. Uma disputa de dados é caracterizada por acessos destrutivos concorrentes, como a escrita em uma variável compartilhada, cujo resultado final não é determinístico. Sua abordagem é baseada em uma centralização dos acessos concorrentes. Cada vez que o programa encontra uma definição, grava uma referência para a mesma em um módulo central. Para cada acesso concorrente com potencial para gerar uma condição de disputa, o programa consulta este módulo central para descobrir se a definição pode gerar um acesso indevido naquele ponto de execução. Apesar deste trabalho ser totalmente focado em paralelização de *software*, contribuiu para o desenvolvimento das definições de alcance mostradas no Capítulo 4.

O trabalho apresentado aqui possui vantagens em relação aos citados nesta seção. Nosso objetivo principal é prover um mecanismo de introspecção seguindo os seguintes princípios:

- O mecanismo **não pode ser intrusivo**. O projetista deve necessariamente não

ter que preparar seu IP para introspecção. Conforme mostra-se no Capítulo 3, na implementação proposta não é necessário nem mesmo o código-fonte do módulo a ser refletido.

- O mecanismo não deve requerer **modificações no ambiente de simulação**. Este trabalho se baseia na biblioteca SystemC padrão, sem modificações. A biblioteca de reflexão utilizada também é de código aberto, permitindo que o usuário altere ou escreva seus próprios módulos sem muito esforço de aprendizado. O ambiente utilizado na aplicação das técnicas apresentadas é o mesmo utilizado somente para simulação, sem necessidade de alteração do compilador, do sistema operacional ou qualquer biblioteca.

- O mecanismo deve ter **pouco impacto no tempo da simulação**. Realizaram-se experimentos (Capítulo 3) que permitem comparar o tempo gasto pela infraestrutura de reflexão com tempo gasto com a adição de um processo SystemC vazio. A implementação proposta também não altera o tempo de simulação interno do SystemC. Quando insere-se o módulo de reflexão na simulação, a penalidade no tempo total gasto na execução da simulação não interfere na temporização interna SystemC, não refletindo portanto na análise temporal dos modelos.

- O mecanismo deve **ser expansível**. Conforme se mostra, derivam-se dois trabalhos diferentes do mecanismo, mostrados no Capítulo 4 e na Seção 3.5.

- O mecanismo **deve prover observabilidade e controlabilidade**. A introspecção deve ser nos dois sentidos (leitura e escrita de valores). Parece óbvio, mas os trabalhos relacionados que utilizam programação orientada a aspectos não conseguem injetar sinais nos módulos.

- O mecanismo **deve proteger a propriedade intelectual** do módulo sob verificação. Os fornecedores de módulos somente devem ser obrigados a revelar informações suficientes sobre o seu produto para permitir sua utilização e verificação.

Capítulo 3

Introspecção em Plataformas SystemC

Neste capítulo, discutimos um mecanismo baseado em reflexão computacional que torna possível que os desenvolvedores interajam com a simulação da plataforma, durante a execução. Este mecanismo permite que os projetistas monitorem e mudem os sinais ou até mesmo os valores armazenados nos registradores internos dos IPs, de forma a explorar, em um alto nível de detalhamento, o que está realmente acontecendo em cada módulo da plataforma. A idéia foi originalmente introduzida em [2], cujo desenvolvimento apresentamos neste trabalho.

Um dos objetivos do que chamamos de metodologia de introspecção é aumentar a capacidade do desenvolvedor de testar os casos extremos, melhorando assim a cobertura funcional das verificações executadas, sem que para isso seja preciso modificar os módulos de descrição de *hardware* ou o código da biblioteca SystemC, bem como preservar a liberdade na escolha do compilador e do ambiente de execução.

3.1 Depuração e Verificação de Modelos em SystemC

SystemC é basicamente composto de um núcleo de simulação e uma biblioteca com estruturas e tipos de dados que representam as estruturas mais comuns em *hardware*, como sinais, portas de comunicação (registradores expostos), vetores de bits, etc. Estes tipos são disponibilizados através de bibliotecas escritas em C++, com a infraestrutura de orientação a objetos normalmente encontrada neste paradigma, como a hierarquia de classes, suporte a heranças e especialização (sobrecarga). Com esta infraestrutura, os usuários de SystemC escrevem os seus próprios modelos, que geralmente são organizados em uma hierarquia composta por vários módulos SystemC interligados entre si de modo a formar uma especificação executável do sistema sendo modelado. Através desta especificação,

espera-se que o desenvolvedor possa simular o sistema descrito, de acordo com o nível de abstração utilizado (ex.: simulação funcional ou comportamental, com precisão de ciclos anotada ou simulada, etc).

Os problemas surgem quando o desenvolvedor precisa depurar um módulo escrito em SystemC com as ferramentas de depuração existentes para C++, como GDB, durante o desenvolvimento do seu módulo de *hardware*. Projetistas de *hardware* não estão interessados nos detalhes internos de implementação do SystemC, mas sim na descrição do modelo que estão trabalhando no momento. As ferramentas para depuração de código C++ não são capazes de distinguir entre a descrição de *hardware* em si e o código interno do SystemC, tornando a depuração uma tarefa árdua e susceptível a erros. Mesmo se compilarmos o SystemC sem suporte a depuração, cada operação onde um dos parâmetros é um tipo de SystemC irá gerar códigos que não interessam ao projetista de *hardware*, que serão interpretados pelo depurador como trechos sem código fonte. A troca de contexto intrínseca do modelo de simulação baseado em eventos usado no SystemC também atrapalha. Ao depurar uma simulação com diversos processos SystemC, o projetista se depara com um único fluxo de execução serializado, sem diferenciação clara entre os processos.

Estas particularidades explicam por que as ferramentas de depuração especializadas em SystemC são praticamente imprescindíveis para que se possa acelerar o ciclo de desenvolvimento dos projetistas de *hardware* que usam SystemC. O problema piora quando temos um projeto baseado no conceito de plataforma, dado que estas podem conter vários módulos complexos em SystemC ligados uns aos outros, incluindo a utilização de módulos específicos para isso. Nestes casos, a tarefa de depuração não se restringe aos módulos isolados, mas também inclui a comunicação entre eles, seja diretamente ou indiretamente, através de uma estrutura de interconexão (um módulo descrevendo um barramento, uma rede, um roteador, etc).

A técnica que apresentamos aqui não só cobre esta lacuna como o faz sem intervenção alguma no código da biblioteca SystemC ou do usuário. Além disso, não é necessário nem mesmo o código fonte do módulo, desde que o arquivo de cabeçalho dos módulos nos quais será aplicada a introspeção seja fornecido junto com o binário correspondente.

3.2 Por que a Reflexão Computacional?

Reflexão computacional é a habilidade de um sistema de observar e modificar a si mesmo. No domínio do *software*, a reflexão é usada para conseguir otimizações de desempenho, por meio de automodificações realizadas pelo programa em tempo de execução, ou para adaptar dinamicamente um sistema a situações específicas. A linguagem Java, por exemplo, possui um mecanismo nativo de reflexão que permite a escrita de um trecho de código genérico, que possa receber instâncias de mais de um tipo de objeto, não conhecidas em

tempo de compilação. Através da reflexão computacional, é possível identificar a classe do objeto em tempo de execução e tomar a decisão de invocar diferentes métodos de acordo com o tipo do objeto recebido.

A reflexão pode ser classificada como estática ou dinâmica, dependendo do mecanismo usado para gerar as estruturas de dados adicionais necessárias à reflexão [10]. A reflexão estática pode ser usada por compiladores para realizar otimizações de memória. Neste tipo de reflexão, o compilador analisa o código em tempo de compilação, gerando estruturas que facilitem o acesso às instâncias dos objetos (ex.: a desalocação da área de memória ocupada por uma instância de objeto, depende da invocação do método de destruição implementado pelo objeto, cujo endereço relativo pode ser calculado por meio de reflexão estática). A reflexão é mais comum na sua forma dinâmica, cujo suporte, quase na sua totalidade, é intrínseco à linguagem. Java, C#, Python, Ruby e Smaltalk são exemplos de linguagens de alto nível com suporte nativo à reflexão computacional.

Mas por que estamos tão interessados em usar a reflexão computacional em um ambiente de desenvolvimento de plataformas de alto nível como SystemC?

Uma das características principais que desejamos na concepção da metodologia de introspecção apresentada neste trabalho é a não intrusividade: os desenvolvedores de IPs não deveriam saber que seus modelos serão inspecionados, significando que deveriam descrever o IP da mesma forma que o fariam se o mesmo não fosse submetido à introspecção.

Mais que isso, queríamos que fosse possível inspecionar, usando o mecanismo de reflexão, os IPs cujo código-fonte possa não estar disponível. Isto parece estranho à primeira vista, porém com a adoção das descrições em alto nível, acreditamos que o mercado de IPs "de prateleira" se desenvolva, tornando possível que um projetista opte por comprar um IP ou parte dele, adequando-o se necessário. De fato, isto vem acontecendo com o mercado de IPs descritos em baixo nível (RTL). *OpenCores* é um repositório de IPs de código aberto, onde se pode encontrar diversos IPs de prateleira, prontos para uso. No caso deste repositório, é possível se obter todo o código fonte, dada a natureza de código aberto do projeto. Porém, em um cenário comercial, é comum que o vendedor não exponha sua propriedade intelectual, fornecendo apenas uma especificação executável do IP, podendo inclusive ser criptografada, e o código fonte da interface para utilizá-lo.

Estes requisitos inviabilizam a utilização da reflexão estática ou qualquer método que necessite de anotação por parte do usuário (onde o mecanismo de reflexão ou o código fonte devem ser reescritos para cada IP). Uma observação importante é que as técnicas de programação orientada a aspectos podem ser uma alternativa interessante para possibilitar a introspecção de dados [7], mas com a desvantagem de que necessitam de uma mudança no paradigma de programação ao qual os desenvolvedores de *hardware* estão acostumados. Cada módulo SystemC no qual deseja-se realizar a introspecção de dados deveria ser desenvolvido seguindo o paradigma de orientação a aspectos. Isto vai totalmente no

caminho contrário do nosso objetivo de minimizar a intrusividade. Além disso, o fato de obrigar os projetistas a gastar um tempo razoável aprendendo e alternando entre dois diferentes paradigmas de programação restringe seriamente a adoção desta alternativa. Pior ainda, todos os IPs existentes teriam que ser reescritos para que a técnica funcionasse.

3.3 Introspecção em Módulos SystemC

Nesta Seção, mostraremos como tornamos possível a introspecção de dados em módulos SystemC através de uma biblioteca externa de reflexão. No nosso caso, escolhemos a biblioteca Reflex-SEAL [29], desenvolvida pelo CERN com parte do projeto ROOT. Os principais aspectos que nos levaram a escolher esta biblioteca são:

- a biblioteca não é intrusiva, não necessitando de nenhuma modificação na descrição do módulo para que a introspecção seja aplicada;
- provê um roteiro de extração de metadados semi-automático, onde um esqueleto da classe do objeto é gerado em uma descrição XML intermediária pelo próprio projetista, através de um programa externo, de código aberto;
- não necessita de dependências adicionais para gerar a especificação executável (exceto pela biblioteca em si);
- não necessita de modificações no compilador e não é dependente de algum compilador específico.

O *software* utilizado para gerar o ambiente de reflexão é composto por:

- GCC 4.4.x
- SystemC 2.2.0 (TLM 2.0)
- Reflex SVN r32289

A Figura 3.1 mostra o fluxo de construção de uma especificação executável, com reflexão habilitada para um módulo SystemC, usando as ferramentas da biblioteca. Os passos necessários basicamente são três: extração de metadados, compilação e ligação (do inglês, *linking*).

A extração dos metadados é feita através de um *script* chamado `genreflex` e é composta de duas fases. A primeira utiliza o GCCXML, um programa de código aberto e parte do conjunto de ferramentas do compilador GCC, para ler o código fonte do arquivo que descreve a interface do objeto, chamado de arquivo de cabeçalho (`.h` ou `.H`, do

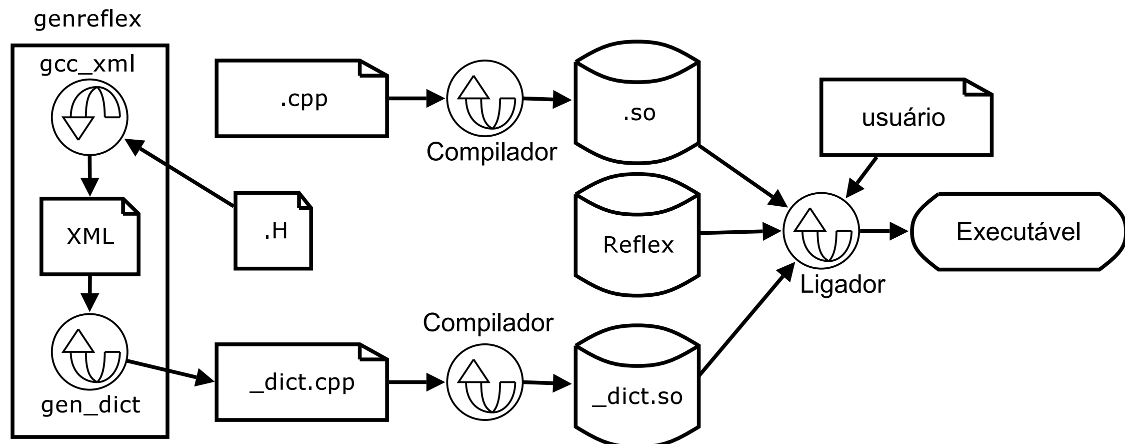


Figura 3.1: Fluxo de reflexão usando a biblioteca Reflex-SEAL

inglês *header*). A saída deste programa é uma estrutura XML equivalente ao objeto e é utilizada sem modificações. Na segunda fase, este arquivo XML é processado e gera-se um arquivo de dicionário compilável, contendo informações sobre a estrutura do objeto refletido, como o endereçamento relativo e tipo dos atributos e métodos. Toda a coleta de informações é feita antes da compilação da simulação e somente precisa ser feita uma vez para cada módulo a ser refletido. Note que o fluxo mostrado na Figura 3.1 não recebe nenhum arquivo de código fonte (.cpp ou .C). Além disso, nenhuma anotação no código se faz necessária para a utilização da metodologia.

O dicionário é importante para os programas que usam os dados refletidos. Usando os métodos disponíveis na biblioteca de reflexão, toda a representação da estrutura do objeto refletido pode ser lida e usada. Normalmente, somente o ponteiro para a instância do objeto a ser refletida é suficiente para as operações de reflexão. Na prática, o que se faz é utilizar o dicionário como uma máscara. Esta máscara representa o objeto em si, portanto a diferença (*offset*) entre o endereço base e qualquer atributo também é a diferença entre o ponteiro da instância e o atributo correspondente. Tendo o ponteiro para para a instância do objeto a ser refletido, podemos aplicar a máscara proveniente do dicionário sobre a região de memória que contém a instância do objeto, de modo a acessá-la em tempo de execução sem que qualquer parte da plataforma tome conhecimento que estamos realizando esta operação. Considerando o cenário de projeto de plataformas, estamos especialmente interessados em inspecionar os atributos dos objetos em SystemC, de modo a permitir sua leitura e escrita. É possível, também, realizar chamadas aos métodos e funções não virtuais do módulo, porém não estamos interessados nesta habilidade aqui.

Após a coleta de informações através da análise da representação intermediária XML, a compilação e ligação seguem os passos que seriam realizados se a reflexão não fosse aplicada. Na Figura 3.1, o código que fará a introspecção em tempo de execução, aplicando a

máscara do dicionário e interagindo com o projetista, está incluído no conjunto de programas inseridos pelo `usuário`, junto com o restante da plataforma. Este programa, assim como o módulo a ser refletido, deve ser ligado à biblioteca de reflexão, ao dicionário de dados e à biblioteca SystemC. O executável produzido na saída é um simulador SystemC da plataforma, capaz de realizar a introspecção para ajudar no processo de depuração e verificação.

SPIRIT é um consórcio criado com o objetivo principal de estabelecer um meio de troca de informações de IPs padrão entre as diversas empresas de EDA. As meta-descrições especificadas pelo padrão SPIRIT permitem que qualquer ferramenta seja capaz de ler de uma forma padronizada todas as informações necessárias para automaticamente importar, exportar e até mesmo compilar qualquer módulo de *hardware* descrito tanto em uma linguagem de baixo nível como em SystemC. Uma das notações padronizadas consistem em uma descrição XML que define *WhiteBoxes* como uma ferramenta de introspecção com capacidade de inspecionar, observar e depurar outros módulos. Apesar de os *WhiteBoxes* serem cobertos na especificação padrão, o consórcio SPIRIT não especifica como os *WhiteBoxes* devem ser implementados, mas sim suas funcionalidades. Sendo assim chamamos nossa implementação de *WhiteBox* inspirada no SPIRIT de *ReflexBox*.

Um módulo de *hardware* escrito em SystemC pode ser visto simplesmente como uma classe escrita em C++. O conteúdo dos registradores, as entradas, saídas e qualquer outro dado que possa ser de interesse para a introspecção de dados são armazenados como atributos de objetos ou variáveis. Este é exatamente o tipo de informação que podemos obter com a reflexão computacional. A implementação do *ReflexBox* consiste em um módulo SystemC que possui um atraso nulo e se comunica com o mundo exterior através de um protocolo simples, expondo um *socket* para conexão. Apesar de ser um módulo SystemC, e se comportar como tal, um *ReflexBox* não pode ser acessado por nenhum outro módulo de SystemC pois não possui interfaces expostas para núcleo de simulação SystemC. Quando o *ReflexBox* é escalonado pelo escalonador do SystemC, utiliza os dicionários disponíveis para aplicar a introspecção sobre a instância do objeto alvo da reflexão (outro módulo de SystemC), de modo que possa realizar as tarefas que a ele forem solicitadas através da conexão externa, um *socket* exposto, onde o usuário deve conectar-se em tempo de execução.

O *ReflexBox* é genérico o suficiente para inspecionar qualquer tipo de módulo SystemC. Cada módulo *ReflexBox* está amarrado a um módulo de SystemC através da instanciação do *ReflexBox*, que leva como parâmetro um ponteiro para o objeto a ser refletido. Um *ReflexBox* pode refletir apenas uma instância de um módulo SystemC, porém uma plataforma pode ter quantos módulos *ReflexBox* forem necessários. Contudo, por motivos óbvios, não é possível refletir estruturas que não são conhecidas em tempo de compilação ou possuem formato criado pelo usuário (ex. uma união (`union`) ou estrutura (`struct`)).

Em nossa implementação, procuramos deixar o código pronto para adaptações caso seja necessário refletir tais objetos.

Apesar de o *ReflexBox* ter a capacidade de refletir métodos, esta característica é pouco utilizada em descrições de *hardware*. É possível invocar qualquer método de uma classe (ex.: uma função em um módulo SystemC), interativamente, em tempo de execução. Esta característica, por exemplo, pode ser usada para inspecionar um registrador criptografado, invocando-se a função decriptográfica sobre o dado contido no registrador durante a depuração do módulo, com o propósito de visualizar o valor descriptografado. A restrição na utilização se deve à necessidade de conhecimento prévio das tarefas que cada método realiza (o *ReflexBox* lista todos os métodos disponíveis), incluindo possíveis efeitos colaterais que o método invocado pode gerar, como mudanças no estado interno do módulo.

Na primeira vez em que um *ReflexBox* é escalonado para execução, gera-se uma lista interna dos atributos refletíveis e permite-se que o usuário informe que atributos deseja refletir no módulo, colocando marcações em cada um. Com isso, o usuário pode selecionar quais atributos são de interesse evitando a reflexão dos que não forem selecionados. Além disso, o usuário pode especificar qual é o interesse, gerando por exemplo um *breakpoint*, que irá parar a simulação quando uma determinada condição for atingida ou até mesmo instruindo o *ReflexBox* a somente registrar qualquer mudança de valor do atributo. Quando uma condição que interrompe a simulação é atingida, o usuário é capaz de observar o valor em questão, interagir com a plataforma alterando seu valor ou solicitar que a execução continue sem alterações. Lembramos que o *ReflexBox* possui um atraso nulo, significando que parar a simulação através dele não avança o tempo de simulação. O escalonamento do *ReflexBox* deve ser feito de uma forma que o mesmo possa inspecionar o módulo refletido a cada ciclo de simulação onde o módulo alvo possa modificar o seu estado. Isto pode facilmente ser conseguido definindo-se a lista de sensibilidade do *ReflexBox* como sendo igual à do módulo a ser refletido, garantindo-se assim que o *ReflexBox* será executado toda vez que o IP também o for.

Na próxima seção apresenta-se um estudo de caso detalhado, ilustrando todos os passos necessários para incluir o módulo *ReflexBox* na simulação, de forma a aplicar a metodologia de reflexão na plataforma e assim tornar possível a sua introspecção.

3.4 Depurando uma Plataforma Exemplo Através de Simulação

A plataforma exemplo roda uma versão modificada do decodificador de MP3 `dist10`. Este código fonte é parte do conjunto de *software* de referência da divisão MPEG da ISO. Foi a última distribuição padrão para o MPEG-1 e MPEG-2, porém suporta MPEG-3 sem

otimizações de desempenho nem qualidade. Usamos este algoritmo por se tratar de um código de referência que pode ser obtido na internet (veja Capítulo A). As modificações que fizemos são: a) o decodificador sempre usa os mesmos nomes de arquivo de entrada e de saída, para simplificar o acesso aos mesmos por parte do processador; b) a cadeia de decodificação MP3 foi dividida em duas partes, de forma que agora é possível rodar o algoritmo em um esquema de *double buffering* usando dois processadores. A estrutura geral do exemplo é mostrada na Figura 3.2.

O núcleo principal de processamento é composto por dois simuladores de processador PowerPC, modelados na linguagem ArchC, cada um executando metade da cadeia MP3. A memória foi particionada em três blocos, permitindo que os IPs sejam mapeados em memória nos endereços entre os três blocos de memória. Entre cada bloco de memória, deixamos propositalmente um espaço de endereçamento em aberto, suficiente para endereçarmos todos os registradores dos IPs.

Os dois IPs implementam funções selecionadas da aplicação original, simulando em *hardware* o que era feito no *software* original. A seleção das funções foi feita por meio de *profiling* do *software* MP3 utilizando o `gprof` [13]. Após rodarmos o *software* na plataforma, usando somente um processador para rodar o algoritmo todo e somente a primeira memória, obtivemos os dados estatísticos do *software* de *profiling*. No caso do desempenho, as melhores funções para serem transformadas em *hardware* são as que consomem o maior tempo de processamento e, dentre estas, as que estão mais perto das folhas na árvore extraída do grafo de chamadas de função. Uma função é considerada folha se ela é chamada por alguma função, porém não chama nenhuma outra. Isso garante que todo o trabalho realizado por esta função é realizado sem dependências externas, tornando-a uma forte candidata a ser transformada em *hardware*. No caso da transformação de uma função não-folha, deve-se optar por transformar todas as funções com chamadas subsequentes em *hardware* ou fazer a troca de contexto entre *hardware* e *software*. O tempo para a troca de contexto não é desprezível e deve ser levado em conta no cálculo do ganho de desempenho proporcionado pela realização da tarefa em *hardware*.

No nosso exemplo, foram selecionadas as funções SBS (*Side Band Synthesizer*) e IMDCT (*Inverse Modified Discrete Cosine Transform*). Estas duas funções somadas representam cerca de metade do tempo de processamento do algoritmo. Como transformaremos estas funções em *hardware*, retiramos as mesmas do *software* original e substituímos por um código que programa e dispara o IP. Programar os IPs significa escrever os valores de entrada nos registradores dos mesmos, o que nesta plataforma nada mais é que escrever em determinadas posições de memória, dado que os IPs são todos mapeados em uma única memória. Os IPs foram construídos de forma a realizarem as tarefas que lhe cabem somente quando receberem uma sinalização para tal, através de um valor pré-estabelecido escrito em registrador específico. Adotamos que um valor zero contido neste registrador

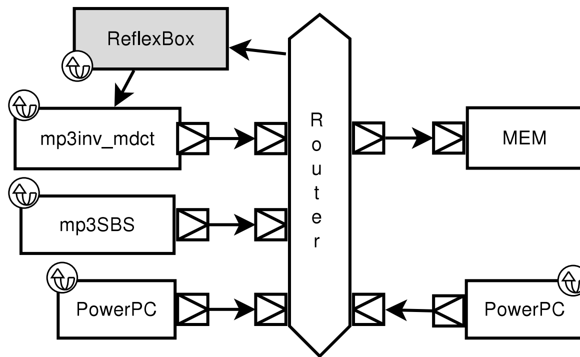


Figura 3.2: Arquitetura da Plataforma MP3 `dist10` modificada

(chamado de registrador de controle) representa que o IP não está trabalhando. Quando o processador escreve um valor diferente de zero neste registrador, o IP lê os valores gravados nos registradores de entrada e começa o processamento. Quando termina, o IP escreve o resultado nos registradores de saída e escreve zero no registrador de controle, sinalizando que terminou a tarefa e está novamente disponível.

No nosso estudo de caso, escolhemos refletir o IP IMDCT. Para isso, temos três passos a seguir:

1. Gerar um arquivo de dicionário para o IP;
2. Instanciar um *ReflexBox*, o módulo que executará a introspecção de dados;
3. Compilar a plataforma para produzir uma representação executável.

Este processo de três passos supõe que a infraestrutura de *software* descrita no início deste capítulo (GCC, SystemC e a biblioteca Reflex-SEAL) já esteja instalada e funcionando. O primeiro passo, gerar o dicionário, é conseguido através da aplicação `genreflex`:

```
1   genreflex ac_tlm_imdct.h -I <path_to_includes>
```

Note que este processo não utiliza o código fonte, somente o arquivo de cabeçalho do módulo, neste caso o arquivo `ac_tlm_imdct.h`. O resultado esperado é um arquivo chamado `ac_tlm_imdct_rflx.cpp`, que representa o dicionário para este IP. Este processo precisa ser repetido para cada IP a ser refletido e pode ser feito pelo fornecedor do próprio IP (neste caso ele deve fornecer o modelo executável do IP e do dicionário, acompanhado do arquivo de cabeçalho para que o projetista possa inserir o IP na sua plataforma). Na Listagem 3.3 pode-se ver uma amostra de uma das classes dentro do arquivo `ac_tlm_imdct_rflx.cpp`. Repare que os membros da classe que representa o IP, `ac_tlm_imdct`, como `target_export`, `answer_ready`, e `memory`, podem ser identificados entre as Linhas 3 – 8.

```
1  ...
2  //------Delayed data member builder for class ac_tlm_imdct
3  void __user__ac_tlm_imdct_db_datamem(Reflex::Class* cl) {
4    ::Reflex::ClassBuilder(cl)
5    .AddDataMember(type_3877, Reflex::Literal("target_export"), OffsetOf(
         __shadow__::__user__ac_tlm_imdct, target_export), ::Reflex::PUBLIC)
6    .AddDataMember(type_14947, Reflex::Literal("memory"), OffsetOf(
         __shadow__::__user__ac_tlm_imdct, memory), ::Reflex::PRIVATE)
7    .AddDataMember(type_2510, Reflex::Literal("answer_ready"), OffsetOf(
         __shadow__::__user__ac_tlm_imdct, answer_ready), ::Reflex::PRIVATE)
         ;
8  }
9  ...
10 namespace {
11   struct Dictionaries {
12     Dictionaries() {
13       Reflex::Instance initialize_reflex;
14       __user__ac_tlm_imdct_dict();
15     }
16     ~Dictionaries() {
17       type_5923.Unload(); // class user::ac_tlm_imdct
18     }
19   };
20   static Dictionaries instance;
21 }
```

Figura 3.3: Trecho do dicionário de dados para o IP `ac_tlm_imdct`

O segundo passo, a instanciação do *ReflexBox*, é realizado no arquivo (`main.cpp`, também conhecido como arquivo principal ou *top-level*). Este arquivo é o local onde o projetista instancia e conecta os módulos da plataforma, na configuração desejada para uma simulação específica (pode-se ter mais de um arquivo principal, representando diferentes configurações da plataforma). Como esperado, aqui também deve-se instanciar o *ReflexBox* responsável pela introspecção em tempo de execução do IP alvo, como ilustrado na Listagem 3.4. A instanciação do *ReflexBox* pode ser feita manualmente para cada IP onde deseja-se aplicar a introspecção, ou automaticamente para todos os IPs da plataforma, postergando a seleção dos módulos refletidos para o tempo de execução.

A Linha 8 contém a instanciação do IP. Para os propósitos deste exemplo, é suficiente saber que este IP é um módulo SystemC qualquer. Na Linha 11, declaramos um *ReflexBox* que está refletirá um IP alvo do tipo especificado pelo nome da sua classe, (`ac_tlm_imdct`), e será responsável pela instância determinada pelo endereço do objeto instanciado na Linha 8 (`&imdct`). O último parâmetro do construtor do *ReflexBox* é um número inteiro representando a porta que o *ReflexBox* deve escutar por uma conexão externa de forma a receber as instruções de introspecção.

```
1  int sc_main(int ac, char *av[]) {
2    sc_clock mclock;
3
4    powerpc ppc_proc1("ppc1"), ppc_proc2("ppc2");
5    ac_tlm_mem mem("mem", 8 * 1024 * 1024);
6    ac_tlm_router router("router");
7    ac_tlm_sbs sbs("sbs");
8    ac_tlm_imdct imdct("imdct");
9
10   //ReflexBox instantiation
11   whitebox<ac_tlm_imdct> reflexbox("reflexbox","ac_tlm_imdct",&imdct
       ,6000);
12
13   char *av1[] = {"dual_ppc.x", "--load=dual_mp3.x", "-A", "source.mp3",
       "dest.aiff", ""};
14   int ac1 = 5;
15   char *av2[] = {"dual_ppc.x", "--load=dual_mp3.x", "-A", "source.mp3",
       "dest.aiff", ""};
16   int ac2 = 5;
17
18   wbmp3sbs.clock(mclock);
19   ppc_proc1.MEM_port(router.target_export1);
20   ppc_proc2.MEM_port(router.target_export2);
21   router.MEM_port(mem.target_export);
22   router.SBS_port(sbs.target_export);
23   router.IMDCT_port(imdct.target_export);
24
25   ppc_proc1.init(ac1, av1);
26   ppc_proc2.init(ac2, av2);
27
28   ppc_proc1.set_instr_batch_size(1);
29   ppc_proc2.set_instr_batch_size(1);
30
31   cerr << endl;
32
33   sc_start();
34
35   ppc_proc1.PrintStat();
36   ppc_proc2.PrintStat();
37   cerr << endl;
38
39   return ppc_proc1.ac_exit_status + ppc_proc2.ac_exit_status;
40 }
```

Figura 3.4: O arquivo `main.cpp` representando a configuração da plataforma MP3 usada no exemplo

O terceiro e último passo é compilar a descrição da plataforma, gerando uma representação executável da mesma. Quando o projetista compila e executa a plataforma, o núcleo de simulação do SystemC instancia e executar cada *thread*, incluindo o *ReflexBox*. Assim que todas as *threads* são inicializadas, o *ReflexBox* irá parar o núcleo de simulação do SystemC e esperar pela conexão do usuário na porta desejada (neste exemplo, 6000). O projetista deverá então conectar-se nesta porta, usando para isso um aplicativo capaz de lidar com conexões via *sockets*, como o `telnet`, ou uma aplicação de depuração compatível com o protocolo (o esqueleto de *ReflexBox* desenvolvido neste trabalho utiliza um protocolo simples, em formato texto, que será explicado adiante). Neste exemplo, utilizaremos uma conexão `telnet` direta, para que o protocolo seja evidenciado. Na Figura 3.5, Linhas 1 – 5, podemos visualizar a conexão ao *ReflexBox* sendo realizada.

```
1  telnet localhost 6000}
2
3  Trying 127.0.0.1...
4  Connected to localhost.
5  Escape character is '^]'.
6
7  G
8
9  user::ac_tlm_imdct::answer_ready;
10 user::ac_tlm_imdct::memory;
```

Figura 3.5: Interagindo com o *ReflexBox* através de `telnet`

Obtendo sucesso na conexão, o projetista pode supor que a plataforma está rodando e a conexão com o *ReflexBox* está feita. Neste ponto, a simulação está pausada esperando por instruções para continuar. O terminal com a simulação rodando irá mostrar algumas mensagens do *ReflexBox*. Neste exemplo iremos explorar algumas funcionalidades do *ReflexBox*, na forma de um tutorial.

A primeira tarefa do desenvolvedor ao se conectar no *ReflexBox* é descobrir o que pode ser refletido. Ao enviar o comando `G` (do inglês *Get*) para o *ReflexBox*, como mostrado na Figura 3.5 (Linha 7), o módulo de reflexão responde com uma lista de todas as estruturas que ele conseguiu refletir, separadas por ponto e vírgula (para uma lista das estruturas refletíveis, vide Seção 3.3). O usuário pode interagir com qualquer um destes itens, como o item mostrado na Linha 9, que representa um registrador.

As últimas duas linhas da Figura 3.5 mostram os dois atributos refletíveis contidos na classe `ac_tlm_imdct`. Os atributos da classe interessantes para a exploração de plataformas são os que representam registradores do IP, como o `answer_ready`. Este registrador indica se o IP está realizando algo útil ou está inativo (valores verdadeiro e falso, respectivamente). Uma introspecção interessante seria um *watch* sobre este registrador, que pára

a simulação sempre que o IP é chamado (este registrador começa com 0, que significa que o IP está, inicialmente, inativo). Para instruir o *ReflexBox* a fazer isso, o usuário deve enviar o comando abaixo através da conexão `telnet`. O comando `I`, quando enviado ao *ReflexBox* pelo usuário, significa a entrada de uma Instrução para o IP, relativa ao objeto refletido. Quando enviado pelo *ReflexBox* para o usuário, representa uma mensagem de Informação.

```
1 Iuser::ac_tlm_imdct::answer_ready;;OnChange;;TRUE;
```

Se tudo estiver correto, o *ReflexBox* irá responder com `Ok`. Se a resposta for qualquer outra coisa, o mais provável é um erro de sintaxe, porém o *ReflexBox* irá imprimir uma mensagem sobre o erro. Neste exemplo, estamos instruindo o *ReflexBox* a monitorar (*watch*) este registrador (`answer_ready`). O próximo passo é reativar a simulação, que se encontra parada pelo *ReflexBox*. Para isso, o usuário insere o comando abaixo, que retira o *ReflexBox* da fase de entrada de comandos e o coloca na fase de simulação. Se o usuário digitar este mesmo comando no início da fase de entrada de comandos (logo após a conexão ser estabelecida), a plataforma irá executar como se o *ReflexBox* não existisse.

```
1 Handshake_Finished
```

A saída do *ReflexBox* na fase de simulação deve ser similar à mostrada na Figura 3.6. O primeiro campo é o tempo de simulação, retirado do núcleo do SystemC. Na primeira linha, o registrador de controle mudou o seu valor para um, indicando que o processador levou exatamente este tempo de simulação para preparar os dados, enviá-los ao IP `ac_tlm_imdct` e escrever o valor um no registrador `answer_ready`. Consequentemente, o IP começará a processar os dados de entrada. Temos que o registrador `answer_ready` mudou novamente de valor em `9821365500ps`. Como esta mudança foi uma mudança de um para zero, podemos concluir que o IP está sinalizando o processador que a computação sobre os dados de entrada foi realizada e os dados estão prontos. A diferença entre estes dois tempos de simulação representa a quantidade de tempo que o IP levou para processar os dados enviados neste ciclo. Usando este *watch*, o projetista pode extrair várias informações, como: quantas vezes o processador invocou o IP para esta entrada, se o tempo gasto pelo processador ou pelo IP é linear ou dependente da entrada, etc.

Watches são bastante úteis, mas vamos tentar um controle maior sobre a simulação. Rodamos a simulação novamente, conectamos via `telnet` e esperamos o *prompt* do *ReflexBox*. Desta vez, vamos digitar os dois comandos da Figura 3.7 e enviá-los ao *ReflexBox*, criando um *breakpoint* condicionado ao valor de um atributo, que representa um registrador.

Com estes comandos, estamos instruindo o *ReflexBox* a parar a simulação quando `answer_ready` for igual a um, e depois terminando a fase de entrada de comandos. A execução continua e pára no mesmo tempo de simulação mostrado na figura anterior, a

```
1  I9819471500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
2  I9821365500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
3  I9821447500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
4  I9823341500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
5  I9823423500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
6  I9825317500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
7  I9825399500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
8  I9827293500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
9  I9827375500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
10 I9829269500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
11 I9829351500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
12 I9831245500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
13 I9831327500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
14 I9833221500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
15 I9833303500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
16 I9835197500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
17 I9835279500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
18 I9837173500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
19 I9837255500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
20 I9839149500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
21 I9839231500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
22 I9841125500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
23 I9841207500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
24 I9843101500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
25 I9843183500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
26 ...
27 Connection closed by foreign host.
```

Figura 3.6: Exemplo de execução do *ReflexBox*

Linha 1 na Figura 3.6.

A simulação está pausada, então podemos interagir com o *ReflexBox* novamente. Vamos modificar o valor do registrador para zero (o que significa que o processador irá entender que o IP acabou o trabalho) e continuar a simulação até o final, mas transformando o *breakpoint* em um *watch*. Esta transformação é necessária, caso contrário não veremos mais o valor do registrador. Para fazer isso, inserimos o comando mostrado na Figura 3.8.

A saída é bem parecida com o que se vê na Figura 3.6, onde tinha-se somente um *watch*, mas o conteúdo final do arquivo de saída, resultado da decodificação do MP3

```
1 Iuser::ac_tlm_imdct::answer_ready;;ConditionEqual;1;TRUE;
2 Handshake_Finished
```

Figura 3.7: Inserindo um `breakpoint` no *ReflexBox*

```
1 Iuser::ac_tlm_imdct::answer_ready;0;OnChange;;TRUE;
```

Figura 3.8: Mudando a condição de parada do *ReflexBox*

pela plataforma, estará errado. O processador entende que o IP acabou o trabalho e lê os dados da saída, que obviamente ainda não está pronta. Para complicar mais, o processador escreverá mais dados na entrada, relativos ao novo pedaço de processamento (segunda rodada), enquanto o IP ainda está processando os dados antigos, levando a uma saída completamente errada. O que queremos chamar a atenção neste exemplo é que o usuário é capaz de modificar os dados durante a simulação. Esta capacidade pode ser uma aliada poderosa quando deseja-se direcionar a simulação para casos de teste extremos ou ambíguos. O projetista pode suspeitar que algo está errado para determinada entrada, então pode forçar que o IP execute sobre ela mesmo que dependa de fatores externos.

Vamos agora tentar identificar um erro na plataforma. Inserimos um erro proposital na plataforma, de forma que a mesma execute até o final sem exceções, porém o arquivo gerado não está correto. Podemos repetir a simulação errônea, porém agora instruindo o *ReflexBox* com os dois comandos da Figura 3.9 na fase de entrada de comandos.

```
1 Iuser::ac_tlm_imdct::answer_ready;;OnChange;;TRUE;
2 Iuser::ac_tlm_imdct::memory;;OnChange;;TRUE;
```

Figura 3.9: Atributos refletíveis do módulo IMDCT, mostrados pelo *ReflexBox*

Estes comandos instruem o *ReflexBox* a criar um *watch* sobre cada um dos registradores `answer_ready` e `memory` (este último é um ponteiro para o local onde os dados a serem processados residem). A saída está na Figura 3.10 (as demais linhas são idênticas ao primeiro exemplo).

```
1 I9819471500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
2 I9819471500 ps;user::ac_tlm_imdct::memory;0x2003EF;FALSE;FALSE;
3 I9821365500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
4 I9821447500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE;
5 I9823341500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE;
```

Figura 3.10: Saída da introspecção do módulo IMDCT

Conforme mostrado, o registrador `memory` mudou de valor em `9819471500ps` junto com o registrador `answer_ready`. Isto aconteceu porquê o processador acabou de programar o IP para sua primeira execução. O problema é que este registrador não mudou mais de valor durante toda a simulação, o que significa que na segunda rodada de execução

(em `9821447500ps`, quando o registrador de controle muda de valor novamente), o IP realizou seu trabalho sobre o mesmo conjunto de dados. Este comportamento não é esperado pois esta plataforma foi implementada usando um esquema de *double-buffering*. Neste esquema, o endereço dos dados de entrada deve ser diferente entre duas rodadas consecutivas.

É claro que o *ReflexBox* não pode identificar erros sozinho, mas ele tem se mostrado uma ferramenta útil, ajudando a descobrir os erros sem muito trabalho. Neste último exemplo, se mudarmos manualmente o valor do registrador `memory` para o valor correto (o endereço do segundo *buffer*), obteremos a saída correta. Isto economiza um tempo considerável na procura do erro no código fonte e em sua correção.

3.5 Verificação de Módulos Isolados

Um dos problemas que enfrentamos quando precisamos simular plataformas é o desempenho dos testes. É sabido que, no desenvolvimento de um novo *hardware*, precisamos de testes em diversos níveis de abstração, desde o funcional até o nível de portas lógicas. Com o advento das linguagens de descrição de sistemas eletrônicos em alto nível de abstração, como o SystemC, abrimos a possibilidade de começar o projeto com uma peça de *software*, chamada de especificação executável do sistema, entrando em um processo cíclico de "refinamento", conforme descrito na Seção 1.

O processo de refinamento consiste em implementar a ideia totalmente em *software*, gerando o que é chamado de especificação executável do sistema, para que seja testada sua funcionalidade, e desenvolver sobre esta implementação o modelo de *hardware*. A partir do *software* testado, desenvolve-se um modelo funcional do *hardware* em uma linguagem de descrição de sistemas eletrônicos (geralmente, SystemC ou SystemVerilog). Esta descrição geralmente é feita usando TLM. Com este modelo em mãos, geram-se quantos modelos forem necessários para o nível de abstração desejado, convergindo para um modelo sintetizável. O processo de transformar um modelo em uma descrição com nível de abstração superior em um modelo com um nível de abstração menor é justamente o refino, que dá o nome à técnica.

A partir da primeira descrição funcional, já é possível integrar este modelo em uma plataforma. O problema aqui é em relação ao desempenho da simulação. Quanto mais detalhado o modelo (menor nível de abstração), mais demorada é a simulação. Na prática, projetistas trabalham com pelo menos dois modelos do mesmo IP, sendo um para integração e simulação comportamental, em alto nível de abstração, e outro para a síntese. Desta forma, os projetistas da plataforma ou de outros IPs da plataforma não precisam se preocupar com o IP em questão, além de contar com um modelo já no início do ciclo de projeto.

Na próxima seção, mostraremos um exemplo de como utilizamos a metodologia de reflexão apresentada na Seção 3.3 para desenvolver uma metodologia que permite acelerar a simulação para os projetistas que estão focados em um determinado IP ou *software*, porém dependem do resultado de outros módulos da plataforma.

3.5.1 *Signal Replay*

Neste experimento nosso objetivo é mostrar como podemos aplicar os conceitos apresentados de forma a prover uma maneira fácil de reutilizar os casos de teste usados em plataformas com alto nível de descrição (incluindo descrições em *hardware*, porém estas com algumas restrições) no ciclo de desenvolvimento. Consideramos que a aplicação em *software* é escrita em C++ e o objetivo final é acelerar a simulação do IP SystemC correspondente, sempre mantendo o modelo correto.

De forma a permitir a reutilização de casos de teste, refletimos as funções e métodos que representam as transações de entrada e saída do módulo, gravamos as saídas e entradas em um arquivo chamado de *trace* e usamos estes dados para imitar a plataforma em simulações posteriores. Chamamos esta abordagem de *Signal Replay*, que funciona como a seguir.

O projetista deve rodar uma simulação considerada correta, que será tida como modelo de referência (do inglês, *golden model*). Nesta simulação, o projetista deve instrumentar corretamente os módulos-alvo (ou o equivalente em nível de abstração diferente) de forma a gerar o arquivo de *trace* para os valores que deseja reinjetar no módulo a ser testado. Caso a simulação de referência seja uma especificação executável do sistema em *software*, deve-se no mínimo ter disponível claramente o particionamento *hardware-software*, para a correta determinação dos pontos de onde se extrai os dados, que representam na especificação a entrada e saída do módulo em *hardware*.

Com este arquivo gerado, o projetista pode trabalhar nos módulos isoladamente usando módulos substitutos, também chamados de *stubs*, para representar o restante da plataforma, rodando efetivamente somente os módulos a serem testados, em uma configuração chamada de "dispositivo sob teste" (DUV, do inglês *Device Under Verification*). Para este tipo de configuração, o arquivo de *trace* deve conter exatamente os valores a serem injetados na entrada do módulo e os valores da saída do módulo, para efeito de comparação. A configuração típica DUV pode ser vista na Figura 3.11.

Esta tarefa pode ser totalmente automatizada usando-se o mecanismo de reflexão *ReflexBox* descrito anteriormente. Se o projetista apontar quais funções, métodos ou atributos representam as entradas e saídas do módulo, podemos dizer que é possível automaticamente gerar um código envoltório que captura os dados da execução em forma de *trace* sem qualquer intervenção do usuário (não intrusivo), e também gerar os módulos

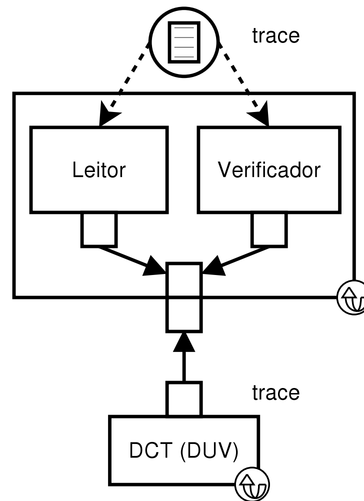


Figura 3.11: Reprodução da simulação do estudo de caso para o *Signal Replay*

substitutos que imitarão a plataforma em simulações posteriores.

Os módulos substitutos, lêem o arquivo de *trace* e alimentam o DUV com os dados de entrada, simulando assim como se o módulo estivesse de fato conectado a uma plataforma. Estes *ReflexBoxes* também recebem a saída do DUV e comparam com a saída gravada no arquivo de *trace* para verificar se a execução está correta. Como a E/S de um módulo em SystemC tem uma assinatura conhecida (parte da descrição) representada tanto pelos atributos do módulo (sinais RTL) quanto pelas chamadas de função (TLM), os módulos substitutos podem ser gerados a partir de um esqueleto usando-se o arquivo de saída do GCC_XML, que é parte da geração do mecanismo de reflexão.

Comparação com *scoreboard*

O *scoreboard* é uma estrutura muito utilizada em verificação para implementar verificações do tipo caixa preta (*black box*), ou seja, cujo objeto de verificação não é exercitado internamente por meios diretos. Nesta técnica, utiliza-se um modelo de referência para gerar respostas para diversas entradas, cuja escolha deve ser maximizada de forma a garantir uma boa cobertura. A escolha dos testes está fora do escopo desta tese (a bibliografia [3] pode ser usada como ponto de partida neste assunto).

Após gerados os estímulos, anota-se em uma tabela, o *scoreboard*, as entradas e as respectivas saídas esperadas. Monta-se uma plataforma de testes similar ao modelo DUV apresentado anteriormente, usando o *scoreboard* como base de consulta para o verificador, como mostrado na Figura 3.12. O injetor é responsável por alimentar o módulo com as mesmas entradas usadas para a geração do *scoreboard*. Na prática, as entradas são

geralmente retiradas da própria tabela. A cada estímulo injetado, o verificador compara a saída do módulo com a esperada (do *scoreboard*).

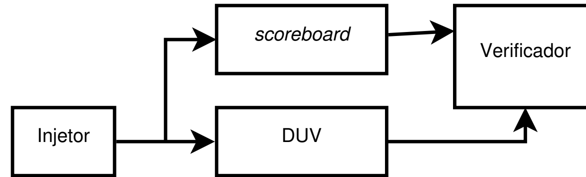


Figura 3.12: Exemplo de Plataforma DUV para *scoreboard*

O *scoreboard* possui semelhança com o *Signal Replay*. De fato, o *Signal Replay* pode ser utilizado como tal. Porém, *scoreboards* possuem observabilidade limitada ao que é exposto nas entradas e saídas do módulo DUV. No *Signal Replay*, qualquer estrutura refletível pode ser verificada, incluindo variáveis representando registradores internos do módulo.

Quando trabalha-se com módulos que possuem algum mecanismo de memória interna, onde os dados de saída não dependem somente da entrada mas sim do histórico da entrada, nota-se claramente a vantagem oferecida pelo *Signal Replay*. No *scoreboard*, necessita-se estimular todas as entradas que fazem parte do histórico, na ordem correta, para que o módulo atinja o estado desejado. Com o *Signal Replay*, pode-se colocar o módulo todo em um determinado estado, com uma tarefa de injeção.

Note que se o *Signal Replay* for utilizado como método caixa preta, o desenvolvedor não terá conhecimento sobre a estrutura interna do IP, porém ainda assim o *Signal Replay* poderá injetar estímulos nas estruturas internas do IP de forma a colocá-lo no mesmo estado que foi capturado na geração do *trace*.

Estudo de Caso para o *Signal Replay*

Nosso estudo de caso é uma plataforma rodando um decodificador MP3 da ISO (*International Organization for Standardization*, uma organização não governamental internacional que publica padrões em diversas áreas) chamado `dist10`[1]. Este decodificador está disponível como software livre para os sistemas operacionais mais utilizados do mercado (ex.: Linux e Windows). A plataforma consiste em um simulador de processador ISS (do inglês, *Instruction Set Simulator*) PowerPC ligado a um barramento com uma memória e outros *IP-Cores* descritos em SystemC. Os *cores* foram escolhidos baseando-se na medição completa do *software* (*profiling*), Sub-band Synthesis (SBS) e a transformada DCT, que são responsáveis por 12% e 37% do tempo de execução, respectivamente. Todas as outras funções permaneceram em *software* executando no processador.

[1]Há um *link* para o código fonte no Apêndice A.

Assumimos um ponto no ciclo de desenvolvimento onde a partição *software-hardware* está clara e os desenvolvedores de *hardware* não estão interessados nos componentes de prateleira, como a memória e o processador, pois podem ser facilmente encontrados no mercado, prontos para integração na plataforma. O foco maior do desenvolvimento são os IPs personalizados, os *cores* SBS e DCT que estão em desenvolvimento, e que são os responsáveis pela maior parte do tempo de execução do decodificador. Além disso, nosso objetivo é que os times que desenvolverão ambos os *cores* possam trabalhar ao mesmo tempo nos seus respectivos projetos, sem interferência mútua.

Após a primeira versão da plataforma executar com sucesso, é possível gerar casos de teste alimentando a plataforma com um arquivo de entrada MP3 e coletar a respectiva saída PCM. Com quatro *threads* de SystemC executando (processador, árbitro de barramento e os dois *cores*), o tempo médio de decodificação para um arquivo MP3 de 6kB é de 41 minutos. Este corresponde a um tempo de simulação impraticável. Os projetistas do módulo DCT poderiam até concordar que este tempo de simulação é aceitável, dado que a tarefa deste módulo contribui com grande parte deste tempo. Porém, os projetistas do módulo SBS certamente não estarão felizes com este desempenho da simulação pois o tempo de simulação deste módulo representa uma parcela pequena do tempo de simulação total da plataforma.

A plataforma foi modelada exatamente como o algoritmo MP3, na forma de um *pipeline*, onde as funções são chamadas em uma ordem específica de forma a processar um pedaço dos dados de entrada. Isto torna difícil a detecção de erros sem possuirmos um arquivo *trace* parcial, contendo no mínimo os dados trafegando nas interfaces dos módulos que fazem parte do *pipeline*. De outra forma, o projetista somente saberá que a falha ocorreu observando a saída do *pipeline*. Para gerar o arquivo necessário, usamos o mecanismo *ReflexBox* para instrumentar os dois módulos e gravar todas transações de entrada e saída nas interfaces dos estágios do *pipeline*.

Com o arquivo de *trace* gerado, foi possível reproduzir o ambiente da plataforma sem que para isso fosse necessário uma execução completa da mesma. Para fazer isso, usamos módulos substitutos baseados no *ReflexBox* para reproduzir os sinais nas interfaces dos módulos, coletar o resultado de uma execução com os sinais injetados e comparar com o arquivo *trace* gerado. Usando esta abordagem, o tempo de simulação foi reduzido para $6,33$ minutos (em média), acelerando a simulação em $7,24$ vezes. A Tabela 3.1 mostra os tempos de simulação medidos para vários fluxos de entradas de áudio MP3.

Os tempos mostrados na Tabela 3.1 são referentes ao tempo médio de 10 execuções não sequenciais, para a plataforma inteira e somente o IP DCT com *Signal Replay*, com o desvio padrão. O primeiro arquivo é um arquivo MP3 nulo (preenchido com zeros, representando a ausência de som). Os arquivos *sweep*, *square* e *noise* são arquivos gerados artificialmente, correspondendo a uma onda de frequência crescente, uma onda quadrada

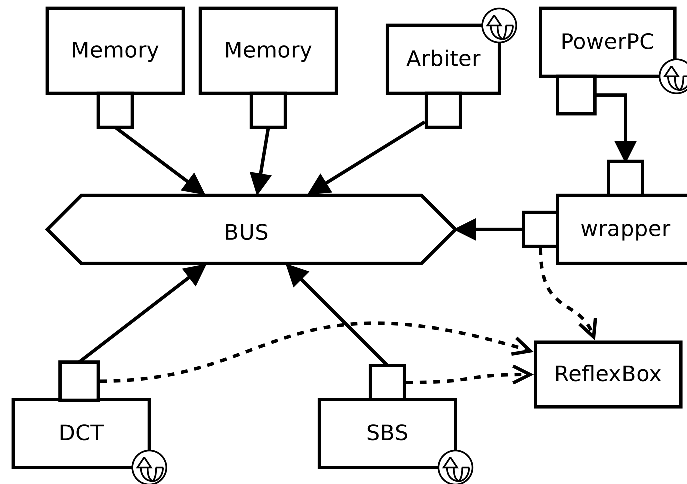


Figura 3.13: A plataforma de estudo para o *Signal Replay* instrumentada usando reflexão

de frequência fixa e ruído aleatório, respectivamente. O último arquivo é um trecho de uma música real. O número após o "b" representa a taxa de bits do arquivo (do inglês, *bitrate*).

Arquivo	**Plataforma**	**DP**	**IP + SR/**	**DP**	**Ganho**
0.mp3 (1.7kB)	672.95	0.17	94.51	0.04	7.12
sweep_b48.mp3 (6.4kB)	2749.22	0.08	379.62	0.13	7.24
square_b48.mp3 (7.0kB)	3753.63	0.30	549.57	0.17	6.83
noise_b96.mp3 (13.0kB)	12294.27	0.34	1743.87	0.24	7.05
mozart_b96.mp3 (12.3kB)	11497.13	0.20	1593.32	0.33	7.22

Tabela 3.1: Tempo médio de simulação (segundos) do MP3 com *Signal Replay*, desvio padrão (DP) e ganho

Para nosso estudo de caso, o arquivo gravado com a E/S para ambos os módulos, quando usando um arquivo de entrada de 6kB, ocupou cerca de 900MB. Considerando que ambos os módulos são relativamente simples quando comparados a módulos IP comerciais, este tamanho pode ser considerado grande, mas como este tipo de arquivo é intrinsecamente um arquivo texto (assim como no nosso caso), um algoritmo de compressão poderia resolver o problema do espaço ocupado pelo armazenamento do arquivo.

A infraestrutura do *Signal Replay* é mostrada na Figura 3.13. O *software* executando no processador é praticamente idêntico ao original. Foram necessárias adaptações somente nos acessos à memória: a plataforma não possui um sistema operacional, portanto não

tem primitivas como `malloc`, transformadas em alocações estáticas de memória. Também incluímos um mecanismo de *double-buffering* para permitir que os dois módulos executem em paralelo. Observe que esta é uma técnica aplicável somente ao domínio do *hardware*, válida somente se os dados da aplicação não sofrem mudanças significativas pois, uma pequena mudança no domínio do *software*, mudará completamente o arquivo de *trace*, invalidando esta técnica. No processo de refino de *hardware*, geralmente executa-se uma simulação correta com uma cobertura de teste suficiente para considerar-se o módulo correto. Esta simulação é usada como referência durante o desenvolvimento e, se gerada corretamente, não sofre alterações, servindo perfeitamente para a aplicação desta técnica.

A redução do tempo de simulação não é a única vantagem do *Signal Replay*. Se um número pequeno de pontos de teste é suficiente para evidenciar um erro, a simulação pode ser abortada sem que seja necessário esperar a execução de todo o *pipeline*. Dissociar a simulação também garante que cada time de desenvolvimento possui uma base de testes completa, impossível de ser invalidada por erros prévios no *pipeline*. Este isolamento pode ser comparado ao conceito de *sandbox* em *software*, onde um programa de computador é executado em um ambiente que restringe as operações que podem ser destrutivas (ex.: entrada e saída, acesso a memória, etc) para poupar o sistema operacional e demais programas executando na máquina de efeitos colaterais causados por erros, propositais ou não, no programa em execução. No caso do DUV, um erro no módulo não será propagado para a plataforma, permitindo assim que times diferentes trabalhem em diferentes partes da plataforma simultaneamente. De fato, um módulo pode ser inteiramente refinado para um nível sintetizável sem qualquer mudança na plataforma.

O *Signal Replay* também se mostrou útil quando fazemos a transição do *hardware* do modelo de referência para o modelo de simulação. O processador usado na plataforma do estudo de caso não possui uma unidade de ponto flutuante. Portanto, a aplicação foi compilada usando uma biblioteca de emulação. Se os arquivos de referência tivessem sido gerados por uma execução da plataforma puramente em *software* e não através da simulação na plataforma, o resultado deveria ter sido adaptado, pois a emulação de ponto flutuante introduz erros de arrendondamento e precisão. Apesar de ainda ser possível que um teste seja bem sucedido, os dados parciais do arquivo de *trace* certamente são diferentes para a execução em *software* e na plataforma pois, durante a descompactação do arquivo MP3 para áudio PCM, os dados intermediários possuem uma precisão maior que os contidos no arquivo final. Uma solução de contorno comum é realizar a verificação dos sinais através do valor RMS e não bit a bit, permitindo um valor de desvio (*delta error*). Esta solução é perfeitamente aceitável, porém introduz mais uma tarefa no ciclo de desenvolvimento.

Nós montamos e testamos com sucesso uma plataforma MP3 partindo de sua implementação em *software* e terminando com um ambiente de *hardware* simulado. Apesar de

ser difícil precisar o tempo que economizamos gerando os ambientes DUV dissociados da plataforma para os dois módulos, quando comparado com uma implementação manual, nós estimamos que a economia é considerável no tempo total de desenvolvimento da plataforma. A redução no tempo de simulação foi expressiva e a sobrecarga inserida pela reflexão é mínima. A técnica claramente não se aplica a plataformas pequenas e para módulos IP que não executam frequentemente. Mas como a experiência nos mostra, tais plataformas e IPs são raros, especialmente quando um processador pode realizar a mesma tarefa (como um *software*-IP).

3.6 Conclusão

Desenvolvemos e aplicamos com sucesso uma metodologia de reflexão capaz de capturar os dados de um módulo IP em uma plataforma de descrição de *hardware* e reinjetar os mesmos dados no módulo, perfazendo assim uma reprodução da simulação de referência, porém com o módulo isolado da plataforma. Mostramos que a sobrecarga da inserção de tal metodologia no fluxo de desenvolvimento é mínima quando comparada com a redução no tempo de simulação. Nossa metodologia pode ser utilizada para ajudar no desenvolvimento de uma plataforma, aplicando a reutilização de casos de teste.

O caminho inverso também é possível. Quando um IP é refinado para a descrição de mais baixo nível, espera-se que o mesmo se torne parte da plataforma numa fase de integração. Isto pode tornar o tempo de simulação da plataforma longo, dado que uma descrição de baixo nível geralmente gasta mais tempo para executar que uma de alto nível. Espera-se que a descrição de baixo nível sirva perfeitamente no lugar da descrição de alto nível. Portanto, a integração deve ser uma mera substituição. Desde que este IP seja considerado correto, pode-se facilmente executar uma simulação completa da plataforma, gravando os dados de E/S para, posteriormente, substituir este IP pelo que chamamos de IP fantasma. Este IP é um módulo vazio cujo comportamento é o mesmo do IP original, porém os dados de saída, ao invés de serem computados, são capturados do arquivo gravado na simulação completa. Esta técnica não é genérica como a descrita anteriormente pois o ambiente de testes supostamente não deve mudar, porém sua utilização pode trazer uma aceleração considerável no tempo de simulação quando comparada a uma execução completa da plataforma.

Apesar de ainda não ter sido testada no desenvolvimento de barramentos, esta técnica encaixa-se perfeitamente neste cenário. O tamanho do arquivo gravado crescerá, pois praticamente todos os dados de entrada e saída passam pelo barramento (senão todas as transações), e estes dados precisam ser gravados. Porém, isto pode ser reduzido usando algoritmos que eliminem a redundância entre casos de teste no arquivo, baseando-se na propriedade que um barramento não deve alterar os dados. É possível até mesmo gerar um

módulo que seja capaz de produzir dinamicamente os dados para alimentar o barramento, garantindo que os valores gerados cubram um subconjunto expressivo de forma a validar o barramento. Este tipo de expansão está entre nossos interesses futuros.

Capítulo 4

Análise de Fluxo de Dados em Plataformas

A análise do fluxo de dados (DFA, do inglês *Data-Flow Analysis*), tem sido extensivamente utilizada na tecnologia de compiladores [1] como uma ferramenta de ajuda na compreensão e otimização de programas. Consiste em determinar e coletar informações sobre a maneira como as variáveis de um programa são definidas e utilizadas. Recentemente, as técnicas de DFA tem sido estendidas para permitir a análise de programas concorrentes [5]. Chamamos a técnica proposta neste capítulo de Análise de Fluxo de Dados em Plataformas (PDFA, do inglês *Platform Data-Flow Analysis*). A PDFA deriva da DFA, combinando a técnica original com técnicas de introspecção de *hardware* (reflexão e sobrecarga de operadores e tipos), para criar um ambiente de análise de plataformas ESL.

4.1 Análise de Fluxo de Dados

Nesta seção, apresentamos sucintamente uma DFA clássica utilizada em compiladores, conhecida como Análise de Definições Alcançantes (do inglês *Reaching Definitions*), a fim de definir as bases para a introdução da PDFA.

Em um programa de computador qualquer, uma **sentença** é a unidade mínima de código capaz de gerar um código equivalente executável, ou seja, gerar uma ou mais instruções na arquitetura alvo, que execute a tarefa representada pela sentença. Na Figura 4.1, mostramos um trecho de um programa em pseudocódigo, com as sentenças numeradas de s_1 a s_{10}. Considere que as variáveis m, n, $u1$, $u2$, $e1$ e $e2$ são variáveis que são válidas e estão disponíveis para uso no início da execução do trecho, sendo as duas últimas variáveis condicionais (verdadeiro ou falso) e as demais inteiros. O código na linha 4, por exemplo, não é uma sentença pois neste exemplo não gera código executável, sendo apenas uma declaração que instrui o compilador a voltar para este ponto (neste tipo de

laço, a próxima instrução) caso a condição da linha 11 não seja satisfeita.

```
1  s1:  inteiro i = m - 1;
2  s2:  inteiro j = n;
3  s3:  inteiro a = u1;
4       faca:
5  s4:     i = i + 1;
6  s5:     j = j - 1;
7  s6:     se e1 entao:
8  s7:        a = u2;
9  s8:     senao:
10 s9:        i = u3;
11 s10: enquanto e2;
```

Figura 4.1: Trecho de código com as sentenças numeradas de s_1 a s_{10}.

```
1  d1: i = m - 1;
2  d2: j = n;
3  d3: a = u1;
4
5  d4: i = i + 1;
6  d5: j = j - 1;
7
8  d6: a = u2;
9
10 d7: i = u3
11
```

Figura 4.2: Definições contidas no trecho de código, numeradas de d_1 a d_7.

Em definições alcançantes, a unidade de informação básica é chamada de *definição*. Uma **definição** é qualquer sentença de um programa que atribua um valor a uma variável: diretamente, através de uma sentença específica para tal; ou indiretamente, através de uma chamada de função ou atribuição a um ponteiro. As definições são identificadas através de uma análise estática do código, atribuindo-se um identificador sequencialmente à ordem de execução.

Esta DFA consiste em determinar as definições que alcançam determinado ponto do programa. No trecho da Figura 4.2, dizemos que `d1` é uma definição alcançante de `d2`, pois após executar `d1` o programa pode alcançar `d2` sem passar por qualquer outra definição da variável `i`. Porém, `d1` não é uma definição alcançante de `d5`, pois `d4` invalida `d1`. Porém, `d4` é uma definição alcançante de `d5`.

Além de uma definição, uma sentença de atribuição pode conter um **uso**. Dizemos que uma sentença usa (ou referencia) uma variável quando esta fizer parte da composição do valor atribuído. No exemplo da Figura 4.2, a definição `d1` define a variável `i` e usa a variável `m`. As constantes são um caso especial, pois se comportam como variáveis imutáveis, ou seja, cujo valor é sempre conhecido e disponível. No decorrer do texto esclareceremos por que ignoramos o uso de uma constante.

A DFA deve coletar as informações sobre as definições em cada sentença do programa. É comum utilizar-se o grafo de controle de fluxo, CFG (*Control Flow Graph*), para restringir o trabalho do analisador implementando a técnica DFA a um bloco básico. Um **bloco básico** é uma sequência de sentenças cujo fluxo de controle inicia na primeira e termina na última, sem possibilidade de parada ou mudança no fluxo de controle, exceto no final. Em um CFG, cada nó do grafo é um bloco básico, ligado por arestas que representam mudanças no fluxo de controle do programa.

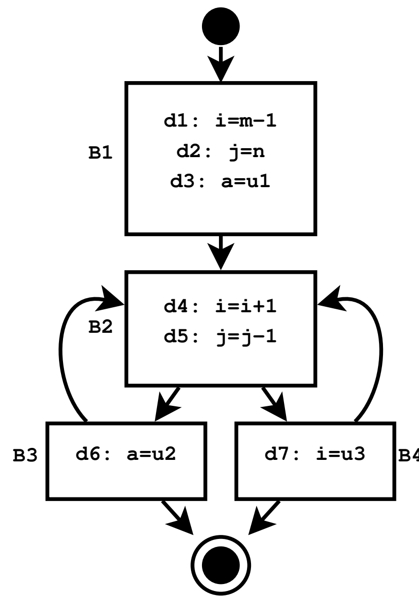


Figura 4.3: CFG para o trecho de código da Figura 4.2

Na Figura 4.3 mostramos o CFG para o trecho de código exemplo, mostrado nas Figuras 4.1 e 4.2. Os blocos básicos foram definidos nas mudanças no fluxo de controle, estão indicados pelos quadrados ao redor das definições e nomeados de `B1` a `B4`.

4.1.1 Conjuntos *generate* e *kill*

Quando uma sentença contém uma atribuição, dizemos que esta sentença gera (*generates*) uma definição. Quando uma sentença gera uma definição que sobrescreve uma definição anterior, dizemos que esta definição invalida (*kills*) todas as definições anteriores da mesma variável. Na Figura 4.2, podemos ver as definições geradas no código da Figura 4.1. A sentença `s6`, por exemplo, não gera definição pois não constitui atribuição de variável, mesmo gerando código executável.

Considerando S um conjunto contendo uma sentença qualquer s, que seja uma sentença de atribuição válida para uma variável a em uma definição d, podemos definir:

$$gen[S] = \{d\} \tag{4.1}$$

$$kill[S] = D_a - \{d\} \tag{4.2}$$

Onde D_a é o conjunto de todas as definições para a variável a do programa.

Na Tabela 4.1 podemos ver os conjuntos *gen* e *kill* para todas as sentenças do trecho de código de exemplo das Figuras 4.1 e 4.2, onde $S = \{s_i\}$. As sentenças com os conjuntos vazios não geram definições, portanto podem ser ignoradas.

Tabela 4.1: O conjuntos *generate* e *kill* para o código de exemplo

Sentença	gen	kill	Sentença	gen	kill
s_1	$\{d_1\}$	$\{d_4, d_7\}$	s_6	$\emptyset$	$\emptyset$
s_2	$\{d_2\}$	$\{d_5\}$	s_7	$\{d_6\}$	$\{d_3\}$
s_3	$\{d_3\}$	$\{d_6\}$	s_8	$\emptyset$	$\emptyset$
s_4	$\{d_4\}$	$\{d_1, d_7\}$	s_9	$\{d_7\}$	$\{d_1, d_4\}$
s_5	$\{d_5\}$	$\{d_2\}$	s_{10}	$\emptyset$	$\emptyset$

Afirmamos anteriormente que as constantes podem ser ignoradas sem prejuízo na DFA clássica, quando resolvendo o problema de definições alcançantes. Assumindo que uma constante é imutável, a atribuição de uma constante deve ser única, então uma sentença que gera uma definição de uma constante, em um programa sintaticamente correto, possui sempre um conjunto *kill* vazio.

Da mesma maneira como definimos os conjuntos *gen* e *kill* para uma sentença, podemos defini-los para um bloco básico. Assumindo que as sentenças de um bloco básico são executadas na ordem que se encontram, o que podemos assumir para a maioria dos programas em linguagens imperativas compilados para arquiteturas não paralelas, podemos definir os conjuntos *gen* e *kill* para um bloco da seguinte maneira:

$$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2]) \tag{4.3}$$

$$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2]) \tag{4.4}$$

Onde S representa a composição de S_1 e S_2, sendo que S_2 é executada posteriormente à S_1. Em outras palavras, o *gen* de uma composição de sentenças é tudo o que é gerado por estas sentenças, sendo válido o último valor gerado. O *kill* pode ser dito como sendo tudo o que ambas as sentenças invalidam, levando-se em conta que, se ambas as sentenças invalidam uma definição, a última é a válida. Aplica-se a composição até que S represente todas as sentenças do bloco básico, partindo-se das Equações 4.1 e 4.2 para as sentenças isoladas.

As Figuras 4.1 e 4.2, possuem um trecho de código cujo CFG é mostrado na Figura 4.3. No CFG, podemos visualizar a divisão em blocos básicos, conforme definida anteriormente. Seguindo as regras das Equações 4.3 e 4.4 para o bloco B_1, começamos definindo os conjuntos *gen* e *kill* para as sentenças individualmente, conforme as Equações 4.1 e 4.2, cujo resultado é mostrado na Tabela 4.1. A segundo passo é compor os conjuntos a partir dos conjuntos para as sentenças, sequencialmente. Temos que, para $S = \{s_1, s_2\}$:

Com S composto, podemos agora compor os conjuntos para o bloco básico B_1, fa-

$$gen[S] = gen[s_2] \cup (gen[s_1] - kill[s_2]) \qquad kill[S] = kill[s_2] \cup (kill[s_1] - gen[s_2])$$
$$gen[S] = \{d_2\} \cup (\{d_1\} - \{d_5\}) \qquad kill[S] = \{d_5\} \cup (\{d_4, d_7\} - \{d_2\})$$
$$gen[S] = \{d_2, d_1\} \qquad kill[S] = \{d_5, d_4, d_7\}$$

zendo a composição das três sentenças pertencentes ao bloco: $gen[B_1] = gen[S \cup \{s_3\}]$ e $kill[B_1] = kill[S \cup \{s_3\}]$. O resultado é mostrado na Tabela 4.2, com os conjuntos para os demais blocos.

Tabela 4.2: O conjuntos *generate* e *kill* para os blocos básicos do exemplo

Bloco	**gen**	**kill**
B_1	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6, d_7\}$
B_2	$\{d_4, d_5\}$	$\{d_1, d_2, d_7\}$
B_3	$\{d_6\}$	$\{d_3\}$
B_4	$\{d_7\}$	$\{d_1, d_4\}$

4.1.2 Conjuntos *input* e *output*

Para alguns tipos de DFA, como a solução do problema de definições alcançantes, derivar informações dos conjuntos *gen* e *kill* facilita a tarefa de visualização e procura dos dados. Os conjuntos *in* e *out* são exemplos de conjuntos derivados do *gen* e do *kill*. O conjunto $in[B]$ é composto por todas as definições que alcançam o início da execução de um bloco básico B. Similarmente, o conjunto $out[B]$ contém as definições que são válidas ao final da execução do bloco básico.

Como base, é seguro afirmar que qualquer sistema computacional possui, em algum ponto considerado a origem da computação, um conjunto *in* conhecido, mesmo que vazio (por exemplo, o início de um programa possui o conjunto *in* vazio e pode ser usado como ponto de partida para o cálculo dos conjuntos do primeiro bloco).

Considerando um bloco básico B, podemos definir:

$$in[B] = \bigcup_{P=predecessor[B]} out[P] \tag{4.5}$$
$$out[B] = gen_B \cup (in[B] - kill[B]) \tag{4.6}$$

Onde P representa todos os blocos que precedem B, portanto $in[B]$ é a união dos conjuntos *out* de todos os blocos que podem executar antes de B.

Os laços devem ser considerados cuidadosamente, pois interferem na definição de predecessor. Na Figura 4.3, B_3 é um predecessor de B_2, pois o laço pode gerar uma condição

onde, quando B_2 iniciar sua execução, B_3 já pode ter sido executado. De fato, isto acontece na segunda iteração do laço, quando o ramo de B_3 for tomado na primeira execução.

Tabela 4.3: O conjuntos *in* e *out* para os blocos básicos do exemplo

Bloco	**in**	**out**
B_1	$\emptyset$	$\{d_1, d_2, d_3\}$
B_2	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_3, d_4, d_5, d_6\}$
B_3	$\{d_3, d_4, d_5, d_6\}$	$\{d_4, d_5, d_6\}$
B_4	$\{d_3, d_4, d_5, d_6\}$	$\{d_3, d_5, d_6, d_7\}$

4.2 Análise de Fluxo de Dados em Plataformas

Descrições ESL são compostas por sistemas com módulos (tanto *hardware* quanto *software*) que, em uma simulação, interagem dinamicamente e em paralelo, durante diferentes ciclos de simulação. Cada ciclo de simulação corresponde a uma unidade de tempo real, definida pelo projetista. Estes ciclos de simulação são divididos em delta ciclos (*delta cycles*), que correspondem a um momento de simulação. Em um delta ciclo, todos sinais envolvidos na simulação são atualizados e em seguida as tarefas de simulação (processos, métodos, etc.) são invocadas. Um ciclo de simulação inicia-se e termina em um estado estável do sistema simulado. O número de ciclos delta para atingir um estado estável não influencia no tempo interno da simulação, apesar de influenciar diretamente o tempo total gasto para simular o sistema.

As sentenças na técnica PDFA não diferem da DFA. Cada trecho mínimo de código da linguagem de descrição de *hardware* capaz de gerar um *hardware* é considerado uma sentença. No caso específico do SystemC, a sentença é exatamente a mesma para a DFA e para a PDFA.

Na técnica PDFA, uma definição deve unicamente identificar uma atribuição de estrutura na plataforma, durante todo o seu histórico de execução. Entende-se por estrutura qualquer *hardware* que pode ser gerado a partir de uma descrição de *hardware* em alto nível. A identificação distingue a PDFA da DFA clássica, pois esta última sempre utiliza informações estáticas. Tendo isto em mãos, podemos definir os conceitos de um *bloco básico* e de uma *definição* no contexto de plataformas.

Definição 1. *Bloco Básico B* Seja I um IP dentro de uma plataforma. Um bloco básico B é composto por qualquer caminho de dados de I que cabe dentro de um delta ciclo de simulação. Se I é um IP de *software* rodando em um processador, o bloco básico B é o bloco básico convencional.

Na DFA clássica, um bloco básico é definido por uma mudança no fluxo de controle de um programa. Na PDFA, um bloco básico é definido pela presença de uma barreira no fluxo de dados de um *hardware*. Em outras palavras, qualquer bloco do IP I composto unicamente por lógica puramente combinacional é considerado um bloco básico. Assim, um bloco básico começa a ser executado sempre que há uma fronteira de dados não ambígua em I (por exemplo, uma escrita de registrador ou uma amostragem de dados). Um bloco básico termina depois da primeira escrita em um registrador, porta de saída ou fronteira explícita de dados em I. A primeira operação do código de um IP I marca a entrada do seu primeiro bloco básico.

A realimentação síncrona de um bloco é feita através de um mecanismo qualquer de armazenamento nas bordas (ex.: registrador), portanto representa o fim de um bloco básico. Porém, a realimentação assíncrona deve ser analisada com cuidado. Cada bloco assíncrono deve conter somente trechos combinacionais não realimentados, considerando-se a realimentação como uma entrada de um bloco e não como parte dele. A técnica não se aplica à circuitos astáveis, ou seja, cujo ciclo de simulação não termina após a execução de um número finito de delta ciclos.

Definição 2. *Definição d.* Uma definição d : (I, a, t) =*operação* em uma descrição de plataforma é uma sentença de atribuição à estrutura de *hardware* representada por a em algum bloco básico de um IP I, no tempo t. Se I é um IP de *software* executando em um processador, d é a definição clássica (estática).

A variável a usada como exemplo na Equação 4.2 na DFA clássica corresponde à estrutura (I, a, t) da definição em plataformas. Na descrição de *hardware* ela representa uma estrutura de *hardware* mesmo que no código da descrição seja uma variável, portanto precisa ser anotada temporalmente. O *hardware* gerado a partir da descrição depende do sintetizador utilizado (um inteiro, por exemplo, pode ser transformado em um registrador).

Definidos os conceitos de definição e bloco básico na PDFA, mostraremos nas próximas seções as definições dos conjuntos *gen*, *kill*, *in* e *out*.

4.2.1 Conjuntos *generate* e *kill*

Assim como na DFA clássica, para resolver o problema *reaching definitions* quando aplicado a plataformas, precisa-se calcular o conjunto de definições geradas e invalidadas para cada bloco básico.

Para uma sentença em uma descrição de *hardware*, a definição dos conjuntos *gen* e *kill* é a mesma da DFA clássica (Equações 4.1 e 4.2):

$$gen[S] = \{d\} \tag{4.7}$$
$$kill[S] = D_{(I,a,t)} - \{d\} \tag{4.8}$$

Onde $D_{(I,a,t)}$ é o conjunto de todas as definições da estrutura a do IP I que já aconteceram, ou seja, com t menor que o tempo atual.

Para compor os conjuntos *gen* e *kill* para os blocos, usamos as mesmas regras da DFA clássica, como vistas nas Equações 4.3 e 4.4. Da mesma forma que na DFA clássica, as regras de composição podem ser utilizadas recursivamente sobre as sentenças de um bloco, até que todas as sentenças do bloco tenham seus conjuntos agregados aos conjuntos do bloco.

4.2.2 Conjuntos *input* e *output*

Não há diferença na definição dos conjuntos *in* e *out* da DFA clássica para a PDFA. Na prática, porém, a PDFA possui uma característica dinâmica que exige o cálculo dos conjuntos a cada nova definição encontrada.

Considerando um bloco básico B, calculamos os conjuntos da seguinte forma:

- para o primeiro bloco básico do IP, $in[B]$ são as definições disponíveis na sua interface de entrada e o $out[B]$ é calculado como na DFA clássica.
- para um bloco cujo *out* já esteja calculado, calcula-se novamente usando as regras da DFA clássica e faz-se a união dos conjuntos (incorporando qualquer nova definição que possa ter sido gerada).

4.2.3 Comentários Sobre a Implementação

Assim como a DFA, a PDFA é um conjunto de técnicas para coletar os possíveis conjuntos de dados calculados em diversos pontos de um sistema computacional que, no caso da PDFA, são simulações de plataformas de *hardware* descritas em alto nível de abstração.

Antes que qualquer técnica de PDFA possa ser executada, o sistema precisa construir os conjuntos de definições. Para o caso do problema de alcance de uma definição, uma definição precisa ser construída para cada operação de atribuição que executa dinamicamente na plataforma. Para conseguir isso, os tipos e operações mais utilizados foram sobrecarregados. No exemplo, nos restringimos aos inteiros e seu derivado em SystemC (`sc_int`), porém a técnica pode ser aplicada a todos os tipos passíveis de reflexão pelo *ReflexBox*.

Na nossa implementação, sobrecarregamos os tipos de SystemC para incluir um marcador único (*tag*) em cada objeto na plataforma. O mecanismo de reflexão computacional descrito no Capítulo 3 foi utilizado para criar todas as definições $d_{(I,a,t)}$, necessárias como entrada para o problema de *reaching definitions*.

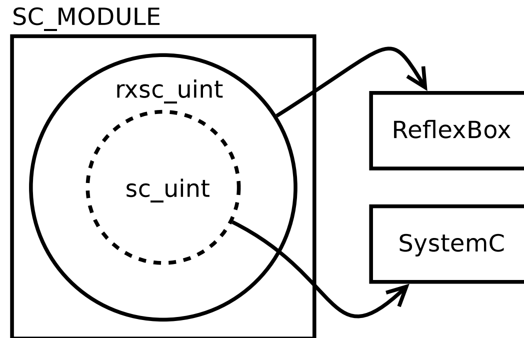


Figura 4.4: Encapsulamento to *tag*

Na Figura 4.4 podemos visualizar o encapsulamento utilizado. Quando um dos tipos sobrecarregados é acessado pelo SystemC, é visto como o tipo original de SystemC, sem o *tag*, pois a sobrecarga mantém todos os atributos originais. Porém, quando o tipo é acessado pelo *ReflexBox*, é possível não só refletir o dado original como também o *tag*, dado que o *ReflexBox* possui o conhecimento sobre a existência do mesmo.

4.2.4 Interações Entre os IPs

Assim como na definição do conjunto *kill*, um bloco básico pode interagir com o mundo externo recebendo definições através de suas portas de entrada. Isto acontece quando uma definição presente no conjunto *kill* de um bloco básico é gerada fora do IP que contém o bloco. Para identificar esta interação, toda vez que uma definição de variável é detectada, todas as definições dentro do mesmo ciclo de simulação são percorridas em busca de outra definição da mesma variável. Se alguma definição for encontrada, esta é marcada como *kill*.

Esta abordagem de rastrear as definições permite que se construa dinamicamente o conjunto *kill* para cada bloco básico de interesse na plataforma, permitindo que o desenvolvedor veja as interações no IP e os seus efeitos colaterais. Observe que este mecanismo é bem genérico e pode ser usado também por módulos de *software*. Apesar desta abordagem não ser capaz de responder se uma interação dentro de um IP está correta ou não, pode facilmente expor as interações, facilitando a depuração da plataforma.

Uma sentença que contém uma utilização de uma variável, no caso de um IP, representa um acesso de leitura a uma estrutura do *hardware* simulado. Como a abordagem

de rastrear as definições computa todas as definições que alcançam uma certa sentença s do IP, permite também que o desenvolvedor faça um busca temporal no histórico das definições que alcançam um uso específico de s. Esta característica pode ajudar na detecção da origem de um problema, principalmente se o problema se manifesta em outro módulo, como mostrado do exemplo a seguir.

4.3 Estudo de Caso #1

Com o objetivo de ilustrar os conceitos apresentados, criamos um estudo de caso, composto por três IPs e uma estrutura de interconexão. Dos IPs, dois se comportam como mestres (ipm_1 e ipm_2) e um como escravo (*ips*). O IP escravo (*ips*) possui três registradores de E/S: um registrador de entrada de dados (`in`), um de saída de dados (`out`) e um terceiro usado para controle (`ctrl`). O fluxo de operação do *ips* é o seguinte: (1) um IP mestre escreve os dados no registrador de entrada do IP escravo; (2) um IP mestre escreve no registrador de controle do IP escravo (`ctrl`), disparando a operação; (3) o IP mestre permanece em um laço de leitura sobre o registrador de controle do IP escravo (`ctrl`) esperando o final da operação; (4) após um período de tempo, o *ips* escreve no registrador de controle (`ctrl`), sinalizando o final da tarefa de execução, quando então o IP mestre pode ler o resultado no registrador de saída do IP escravo (`out`).

Na Figura 4.5 mostramos o grafo de fluxo de controle (CFG, do inglês *Control Flow Graph*) para um IP mestre e um escravo, usando um pseudocódigo extraído do código SystemC original. Os IPs podem também ser IPs de *software*, caso em que a estrutura dos blocos básicos deve seguir a definição apropriada como na Definição 1. O código para os dois IPs mestres é idêntico, dado que os mesmos são instâncias do mesmo IP. Os blocos básicos são numerados como $B_k, k = 1, 2, \ldots$.

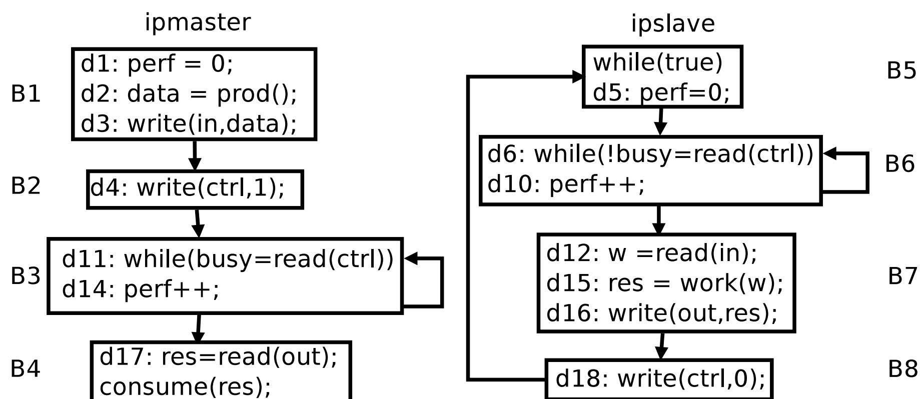


Figura 4.5: CFG do estudo de caso #1

A numeração das definições segue a ordem de execução dentro da plataforma, pois elas são geradas dinamicamente. Omitimos as definições do ipm_2 pois são as mesmas do ipm_1, porém com um ciclo de diferença (atrasadas).

Supomos que `write(a,b)` é uma função sem efeitos colaterais, que escreve o valor `b` no endereço `a`. Similarmente, `read(a)` retorna o dado armazenado no endereço `a`. Estas funções correspondem às operações de leitura e escrita externas do IP. O leitor deve interpretar `write(a,b)` como uma operação de ES que envia o valor local de `b` para uma porta/registrador de saída `a` externa ao IP.

Na Figura 4.5, o bloco básico B_1 do módulo mestre ipm_1 contém todas as sentenças desde o início do código até o primeiro acesso externo do IP (inclusive), que marca o final de B_1, de acordo com a Definição 1. O próximo bloco (B_2) é composto somente por uma operação `write` (d_4) pois, assim como antes, esta operação denota um acesso externo. B_3 começa logo após B_2 e vai até o ponto imediatamente antes do primeiro acesso a um registrador externo (ex. `out`) em d_{17}. O bloco B_4 começa em d_{17} e vai até o final do código do IP. O código do IP escravo pode ser dividido em blocos básicos de forma similar.

Na Figura 4.6, mostramos todas as definições que foram capturadas após cinco ciclos de simulação da plataforma exemplo. As definições são mostradas na ordem em que foram capturadas.

Este exemplo revela um problema de concorrência típico, que pode potencialmente ocorrer em qualquer plataforma paralela. O problema acontece devido ao uso concorrente do IP escravo pelos dois IPs mestres. Assumimos aqui que o desenvolvedor da plataforma não previu a utilização de nenhum mecanismo apto a lidar com os problemas de recursos compartilhados, ou até pior, assumiu que o *ips* estava preparado para lidar com acessos concorrentes. Sendo assim, mesmo que cada IP esteja individualmente correto, a combinação dos mesmos na plataforma pode produzir um comportamento incorreto, como descrito abaixo.

Na Figura 4.6, os dois mestres esperam pelo resultado em um laço de espera (do inglês, *busy-wait*), amostrando o registrador de saída do *ips* quando o registrador de controle indicar um estado de "pronto", representado nesta plataforma pelo valor `0`. Os mestres começam a executar com um ciclo de diferença entre eles: ipm_1 escreve no registrador de entrada de *ips* (d_3) no ciclo $t1$, disparando a execução do IP escravo no ciclo $t2$, enquanto ipm_2 faz a mesma coisa, porém um ciclo atrasado, em $t2$ (d_9).

No ciclo $t3$, o resultado da computação de *ips* está pronto, disponível no registrador de saída, e o valor do registrador de controle é escrito com `0` para sinalizar isso (d_{18} em $t4$). O IP mestre ipm_1 detecta esta mudança do registrador de controle através da sua leitura (d_{11}) e sai do laço de espera. O problema é que no ciclo $t2$, o IP ipm_2 também escreveu no registrador de entrada de *ips* (d_9). Desta forma, ipm_1 amostra um resultado incorreto do registrador de saída de *ips* (d_{17}), pois este valor foi computado usando o

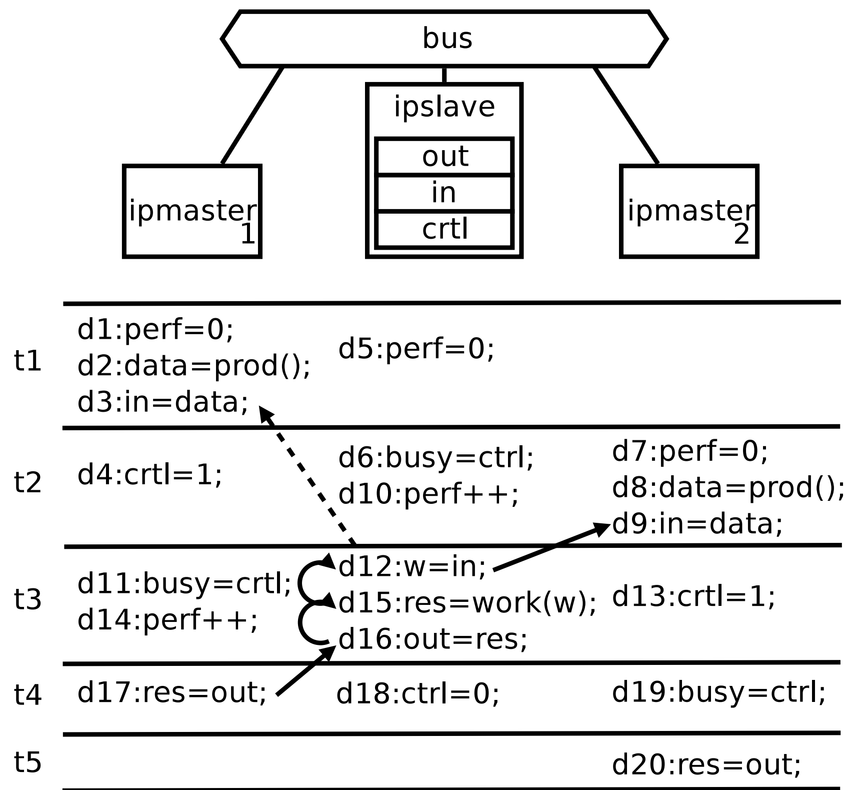


Figura 4.6: Plataforma do estudo de caso #1

dado de entrada fornecido por ipm_2. Indo mais além, ipm_2 provavelmente amostrará o resultado corretamente. Como consequência, a operação desta plataforma é fortemente dependente da sua temporização. Se os valores forem escritos e amostrados de forma que o sincronismo entre os IPs nunca se sobreponha, a plataforma funcionará corretamente.

O desenvolvedor pode contar com a técnica PDFA para detectar este tipo de problema. Isto pode ser conseguido usando *reaching definitions* para construir a cadeia uso-definição, ou cadeia-ud (ud-chain, do inglês *use-define chain* [1]) para a sentença associada a definição d_{17}. Para fazer isso, devemos primeiro calcular os conjuntos *generate* e *kill*.

Na Tabela 4.4, mostramos os conjuntos *gen* e *kill* construídos em tempo de execução usando o mecanismo de reflexão. As *reaching definitions* são então resolvidas conforme descrito anteriormente, resultando nos conjuntos *in* e *out* dos blocos básicos. Depois que $in[B_4]$ é determinado no ciclo $t4$, a variável `out` de d_{17} pode ter sua cadeia-ud calculada: $d_{17} \mapsto d_{16} \mapsto d_{15} \mapsto d_{12} \mapsto d_9$. As setas conectando as definições na Figura 4.6 mostram a cadeia-ud com sua sequência de dependências.

Suponha agora que o desenvolvedor descubra que o valor de `out` na definição d_{17} está errado (Figura 4.6). Para detectar este erro, podemos atravessar a cadeia-ud ao contrário, até que atinjamos o final da sequência em d_9 no ipm_2. Portando, o valor de `out` em ipm_1

Tabela 4.4: O conjuntos *generate* e *kill* para o estudo de caso #1

Bloco	**gen**	**kill**
ipm_1		
B1	d_1, d_2, d_3	d_9, d_{14}
B2	d_4	d_{13}, d_{18}
B3	d_{11}, d_{14}	d_1
B4	d_{17}	
ipm_2		
B1	d_7, d_8, d_9	d_3
B2	d_{13}	d_4, d_{18}
B3	d_{19}	
B4	d_{20}	
ips		
B5	d_4	d_{10}
B6	d_6, d_{10}	d_4
B7	d_{12}, d_{13}, d_{16}	
B8	d_{18}	d_4, d_{13}

está sendo calculado usando o valor `in` fornecido pelo ipm_2 em d_9. Isto foi indicado pelo fato de d_3 ser invalidada por d_9, não alcançando d_{17}. O comportamento esperado é representado pela seta tracejada.

Esta situação potencialmente anormal pode ajudar o projetista a descobrir a origem do valor errôneo de `out`. Embora seja possível detectar este tipo de anomalia automaticamente, a análise final deve ser deixada ao projetista pois requer conhecimento sobre a plataforma. Neste caso, deveria-se ter a informação que o IP usando o escravo quer o seu resultado calculado sobre os dados de entrada que ele mesmo forneceu, não sobre os dados gerados por outro IP. Não é possível inferir este tipo de informação, pois esta característica de concorrência pode ser proposital ou desejável pelo projetista.

4.4 Estudo de Caso #2

O segundo estudo de caso mostra como a PDFA pode ser usada como uma ferramenta para extrair o paralelismo durante a exploração do ambiente de desenvolvimento.

A aplicação deste exemplo é baseada na implementação padrão em *software* do algoritmo de *hash* SHA1, proveniente do conjunto de testes Mibench [14]. Este algoritmo

é composto por um cálculo em 80 passos, dividido em 4 funções, sendo cada uma das funções aplicadas ao fluxo de entrada 20 vezes. A entrada é composta por pedaços de 521 bits e a saída é uma palavra de 160 bits.

Nossa exploração começa com uma plataforma minimalista, composta por um barramento, um processador e um IP, contendo toda a aplicação SHA em *software*. O processador é responsável por gerar o fluxo de entrada e alimentar o IP, que é responsável por todo o resto. De acordo com a definição de bloco básico na Seção 4.2, este IP é composto somente por um bloco básico, dado que o mesmo lê a entrada no início da execução e escreve na saída somente quando toda a computação foi completada.

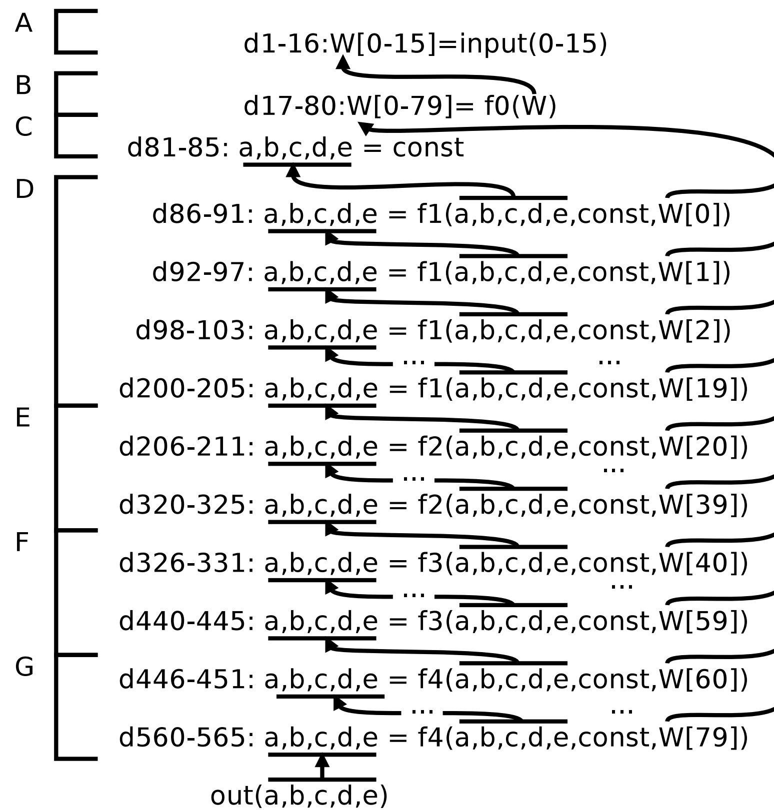


Figura 4.7: Organização interna da primeira versão do IP SHA

A exploração do ambiente de desenvolvimento procede como a seguir. Primeiro, o IP é executado para que as definições sejam capturadas como foi explicado anteriormente. As definições resultantes são mostradas na Figura 4.7, divididas em blocos funcionais, nomeados de `A` a `G`. O bloco `A` faz somente a cópia da entrada (16 palavras de 32bits) para o vetor `W`. O bloco `B` gera as demais 64 palavras baseado em `W`, completando as 80 entradas necessárias para o algoritmo (as primeiras 16 são mantidas intocadas).

Até este ponto, o conjunto *out* do IP consiste em 80 definições, d_1 a d_{80}. Uma primeira

Tabela 4.5: Conjuntos *in* e *out* para o estudo de caso #2

Bloco	**in**	**out**
A		$d_1 - d_{16}$
B	$d_1 - d_{16}$	$d_1, d_2, ..., d_{80}$
C	$d_1 - d_{80}$	$d_1 - d_{85}$
D	$d_1 - d_{85}$	$d_1 - d_{80}, d_{200} - d_{205}$
E	$d_1 - d_{80}, d_{200} - d_{205}$	$d_1 - d_{80}, d_{320} - d_{325}$
F	$d_1 - d_{80}, d_{320} - d_{325}$	$d_1 - d_{80}, d_{440} - d_{445}$
G	$d_1 - d_{80}, d_{440} - d_{445}$	$d_1 - d_{80}, d_{560} - d_{565}$

análise revela que as definições em *out* alcançam o final da execução do IP sem serem modificadas. Isto significa que nenhuma palavra gerada no final do bloco `B` é invalidada até o final do IP. Além disso, a cadeia-ud mostra que todas as 80 definições dependem somente da entrada e que cada uma delas é usada somente uma vez durante todo o ciclo de vida do IP. Mostramos os conjuntos *in* e *out* para cada um dos blocos básicos na Tabela 4.5.

A saída do IP é exatamente o conteúdo dos registradores `a`, `b`, `c`, `d`, e `e`. Se o desenvolvedor escolher qualquer um dos usos destas variáveis (incluindo a escrita para a saída no final do IP) e atravessar inversamente a cadeia-ud, ele sempre irá parar em uma das definições dentro de `C` porque este bloco mata todas as definições prévias destes registradores usando constantes, quebrando assim a cadeia-ud. Isto nos mostra que cada execução do IP não depende de execuções anteriores, significando que o desenvolvedor pode incluir na plataforma quantos clones deste IP ele desejar, processar cada pedaço da entrada separadamente em um dos IPs e juntar o resultado no final. Esta é exatamente a conclusão mostrada em [16], usando uma abordagem *ad-hoc*, onde os autores paralelizaram o SHA1 clonando o IP inteiro. Mostraremos a seguir como a PDFA pode ser usada para extrair ainda mais paralelismo, além da técnica de clonagem do IP mostrada em [16].

A computação principal do IP é executada pelos quatro blocos nomeados de `D` a `G`, que representam exatamente as quatro funções de grupo do algoritmo SHA. Os blocos diferem em dois pontos: a função que cada bloco aplica sobre os dados e a constante usada na computação. No entanto, cada bloco tem sua própria função e sua constante, e estes não mudam durante as vinte iterações de cada bloco. Na Figura 4.7, cada linha representa uma execução do bloco. O bloco `D`, por exemplo, gera as definições d_{86} a d_{91} na sua primeira execução, d_{92} a d_{97} na segunda execução e assim por diante, até a última execução deste bloco, que gera d_{200}a d_{205}.

É importante salientar que cada definição mata a definição correspondente do ciclo

de execução anterior, assim d_{92} mata d_{86}, d_{93} mata d_{87}, e assim por diante. Isto nos leva aos conjuntos *out* mostrados na Tabela 4.5. O resultado da execução do bloco `D` são as definições d_{200} a d_{205}, mais o conjunto *out* do bloco `B`, intocado. O mesmo comportamento pode ser observado nos outros três blocos (`E`, `F`, e `G`).

As cadeias-ud do registradores de saída dentro dos blocos `D` a `G` mostram que esta parte do código não pode ser paralelizada. O resultado final para o registrador `a`, por exemplo, é a definição d_{565}, que depende de d_{560}, d_{559}, d_{554} e assim por diante, até alcançarmos d_{81} no início do bloco `C`. Esta cadeia pode ser vista na Figura 4.7, começando do registrador `out` no final do IP e rastreando até d_{81} no bloco `C`.

Apesar desta cadeia não ser paralelizável, ela sempre começa no bloco `C` e termina no bloco `G`. Além disso, cada definição para as variáveis `a` a `e` mata uma definição anterior (exceto pelo começo da cadeia no bloco `C`). Este comportamento é exatamente o comportamento de uma estrutura de IPs encadeados, então a cadeia-ud em questão pode ser paralelizada na forma de um *pipeline*.

Considerando a análise feita acima, construímos a plataforma mostrada na Figura 4.8, extraindo os blocos de `D` a `G` como uma cadeia de IPs encadeados.

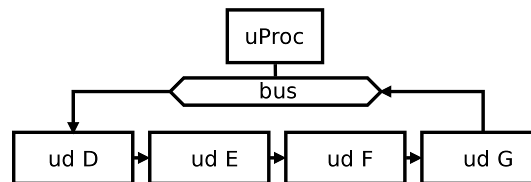


Figura 4.8: A plataforma SHA

Neste modelo, colocamos todo o trabalho nos blocos de `A` a `C` dentro do processador, dado que a tarefa delegada a esta parte é executada somente uma vez a cada porção de dados a ser processada. O processador deve calcular o vetor `W` e enviá-lo para o IP correto. Se rodarmos uma análise de uso sobre todo o vetor `W` (d_1 to d_{80}), podemos facilmente descobrir que cada posição do vetor é usada somente uma vez em toda a execução do módulo. Mais ainda, as definições de d_1 a d_{20} são usadas somente pelo bloco `D`, as próximas 20 definições pelo bloco `E`, e assim por diante. Quando o processador dispara a execução do bloco `D`, ele já deve escrever os valores de `W[1]` a `W[19]` no IP que representa este bloco. Após 20 ciclos de execução, este *core* termina sua computação (quando d_{205} é executada), e os próximos valores (de `W[20]` a `W[39]`) devem estar no IP que representa o bloco `E` de forma que este IP tenha os dados necessários para começar o seu trabalho. Desta forma, o processador pode utilizar este período de 20 ciclos de execução para fazer outra coisa, como por exemplo gerar estes valores.

Outra otimização importante pode ser inferida a partir do conjunto *kill*. O conjunto *kill* para o bloco `D` inclui as definições d_{206} a d_{211}, que foram geradas na primeira execução

Tabela 4.6: Tempos de execução para o estudo de caso #2

Refletindo		Execução	Tamanho do *Trace*	Análise
1	Nada	17m32.003s	-	-
2	D	17m56.412s	3.6MB	0m12.574s
3	E	17m57.039s	3.1MB	0m18.023s
4	D,E	18m02.654s	6.9MB	3m37.564s
5	Tudo	18m08.289s	16.3MB	9m56.878s

do bloco `E`. Isto significa que, uma vez que o bloco `E` termina sua primeira execução, todas as definições pertencentes a `D` estão invalidadas. Observe que neste exato instante, já é possível inserir outro pedaço de dados a ser processado no IP representando o bloco `D`, mantendo o *pipeline* cheio. Sendo assim, uma vez que o *pipeline* esteja cheio, o processador pode amostrar um pedaço de dados já processados pela cadeia na saída de `G` a cada 20 ciclos de execução, ao invés de esperar o processamento original de 80 ciclos.

4.5 Experimentos

Coletar as definições para a técnica de PDFA é uma tarefa primordial, que necessita ser feita de uma forma rápida e com o mínimo de intrusão.

Tanto a reflexão quanto a sobrecarga são tarefas bastante rápidas, constituindo um tempo de simulação adicional negligível. As tarefas que consomem mais tempo nesta técnica são o *logging* (escrita dos dados coletados em memória permanente) e a análise dos dados coletados.

A Tabela 4.6 mostra alguns exemplos de temporização medidos na plataforma do estudo de caso #2, apresentada na Seção 4.4. A primeira linha da tabela mostra a execução da plataforma sem nenhuma instrumentação. A linha 2 mostra os tempos de execução para a plataforma, porém agora com o IP `D` sendo refletido para a coleta de dados. O mesmo é aplicável para o IP `E` na linha 3. A linha 4 mostra os números quando ambos os IPs `D` e `E` são refletidos. Finalizando, a linha 5 mostra o desempenho quando aplicamos a reflexão sobre todos os quatro IPs da cadeia.

Cada execução foi rodada usando o mesmo arquivo de entrada, gerado a partir de 2MB de dados pseudo aleatórios, e a saída foi comparada com uma implementação em *software* (modelo confiável, do inglês *golden model*). Todos os testes foram executados usando uma máquina Intel P8400 Core2Duo com Gentoo Linux e 4GB de memória. Os tempos representam a média entre 10 execuções não consecutivas.

Pelos números na Tabela 4.6, pode-se notar que refletir um IP ou até mesmo a plata-

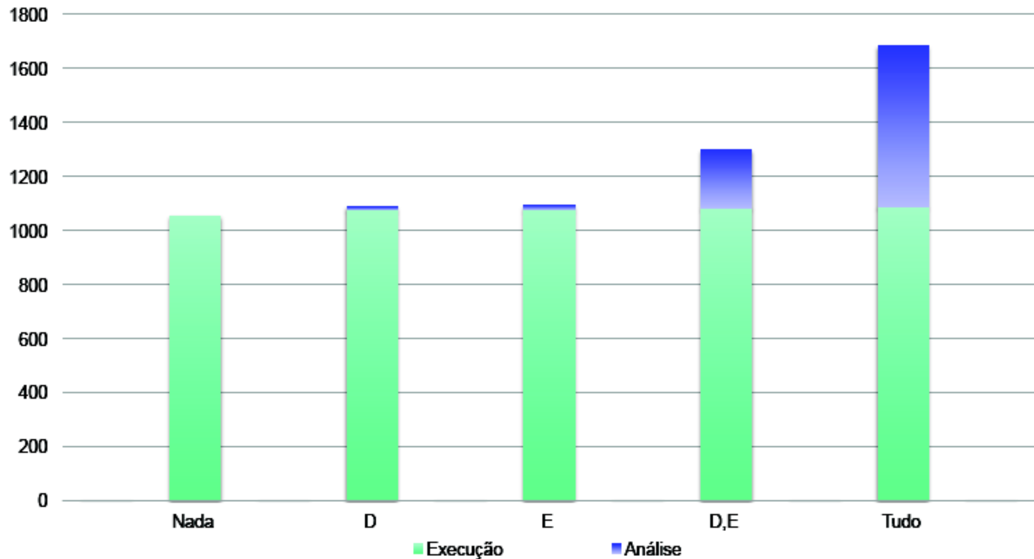


Figura 4.9: Tempos de execução em segundos para o estudo de caso #2

forma toda não gera um impacto considerável no tempo de execução da plataforma. O tamanho do arquivo de *trace* é proporcional ao número de definições presente em cada IP. O maior impacto é a análise do arquivo de *trace*, que toma a maior parte do tempo gasto e cresce rápido quando temos interações entre os IPs (devido ao fato que uma definição de um IP pode matar uma definição em outro IP). Note que, na coluna `Análise`, nas linhas 2-3 da Tabela 4.6, refletindo somente um IP não cria muita sobrecarga no tempo, pois as definições deste IP não são invalidadas por definições externas. Quando há interação entre os IPs, a análise do *trace* se torna uma grande consumidora de tempo, assim como mostrado na linha 4, coluna `Análise` da tabela. O gráfico da Figura 4.9 mostra os tempos de execução da simulação e análise, somados em cada barra de forma a obtermos o tempo total gasto com a aplicação da técnica.

Em geral, instrumentar a PDFA na plataforma não produz nenhuma sobrecarga extra, quando comparada com outras formas conhecidas de técnicas de introspecção, com a vantagem de oferecer aos projetistas o ferramental de análises mostrado.

4.6 Conclusão

Neste capítulo, introduzimos uma técnica de análise de fluxo de dados aplicada a plataformas de descrição de *hardware*, que chamamos de PDFA (do inglês, *Platform Data-Flow Analysis*), uma abordagem que combina reflexão computacional, sobrecarga e a análise clássica DFA (do inglês *Data-Flow Analysis*), comumente aplicada a *software*, de forma

a prover ao desenvolvedor uma ferramenta de análise. Especificamente, as contribuições principais deste capítulo foram: (a) uma abordagem de rápida implementação e não intrusiva de marcação das definições de uma plataforma, no domínio temporal; (b) uma técnica para coletar os dados necessários e resolver o problema de *reaching definitions* dinamicamente, no nível de plataforma, e construir as cadeias-ud que podem ser usadas para verificação e otimização.

A técnica PDFA proposta tem potencial para abrir novas maneiras de se observar e analisar uma plataforma de descrição de *hardware*. A principal razão para afirmarmos isso é a generalidade da técnica, pois esta pode ser aplicada para IPs tanto no domínio do *software* quanto do *hardware*, criando uma visão uniforme da plataforma. Outras técnicas de análise além do problema de *reching definitions* estão em estudo, as quais podem expandir a aplicabilidade da PDFA para outras tarefas de verificação e otimização de plataformas.

Capítulo 5

Conclusão e Trabalhos Futuros

Migrando as descrições de *hardware* de um nível baixo, próximo do silício, para um nível alto, próximo das linguagens de programação modernas, os projetistas de *hardware* podem se beneficiar das ferramentas de projeto existentes para o desenvolvimento de *software*. As descrições ESL possibilitam uma integração precoce de *software* e *hardware*, possuem inúmeras ferramentas de depuração (a maioria herdada ou adaptada do domínio do *software*) e ainda geram modelos que são mais rápidos de simular e são mais fáceis de depurar e corrigir que os equivalentes em descrições de baixo nível. Porém, estes ganhos possuem um preço. O aumento da complexidade dos sistemas eletrônicos é advindo da necessidade evolutiva intrínseca do cada vez mais exigente mercado, não tendo relação direta com a adoção de descrições de alto nível. Além da degradação na precisão da caracterização de potência e latência, o preço a que nos referimos é um aumento na complexidade da descrição em si. É claro que a complexidade nos leva a uma necessidade de ferramentas mais completas e automáticas, porém ressaltamos as desvantagens inerentes de uma mudança de paradigma. Inserimos no fluxo de projeto bibliotecas e padrões inexistentes ou incomuns, como o SystemC e a reutilização de IPs, nas descrições de baixo nível. Problemas como o casamento da assinatura de um módulo simplesmente não existiam até o aparecimento do SystemC. Além disso, as ferramentas usadas no domínio do *software* nem sempre são adequadas para a utilização no domínio do *hardware*.

A depuração de módulos é uma tarefa árdua se comparada à depuração de um *software* puro, porém houve avanço em relação à depuração no nível de bits, tanto no tempo dispendido quanto à propensão a erros. Visando facilitar a depuração, propusemos a reflexão como caminho para a introspecção de plataformas. Esta metodologia não visa substituir a depuração ponto a ponto oferecida pelas ferramentas de *software* como o GDB, porém alia-se a esta técnica como um filtro e como base de apoio para a leitura e escrita de dados não intrusivos em uma plataforma. Mostramos no Capítulo 3 como montamos o arcabouço de reflexão e como utilizamos esta metodologia para desenvolver

o *ReflexBox*, um módulo de SystemC especializado em introspecção de módulos SystemC especialmente útil em plataformas. Mostramos ainda um exemplo da utilização deste módulo sem intrusão alguma no código do usuário, possibilitando que seja usado para depuração em IPs cujo código fonte não possa ser obtido.

Com a base desenvolvida pelo *ReflexBox* e a metodologia de reflexão, pudemos evoluir a técnica para acelerar a verificação de módulos isolados. Na Seção 3.5 mostramos a aplicação de um *ReflexBox* modificado como base para a extração e injeção de dados em um DUV, onde pudemos imitar uma plataforma SystemC do ponto de vista do módulo em teste. A técnica baseia-se em uma execução controlada e monitorada da plataforma como um todo, com instrumentação dos pontos chaves para a criação do modelo DUV. Após isto, montamos a estrutura de entrada e saída e colocamos o módulo alvo no centro, injetando os sinais do arquivo gravado da execução padrão da plataforma no módulo. A saída é comparada com a mesma saída obtida na execução padrão, possibilitando uma verificação por comparação. Para o refinamento do nível de abstração de módulos SystemC, a técnica se mostrou aplicável. Mostramos ainda um exemplo onde evidenciamos que o ganho no tempo de simulação chega a 7.6 vezes, permitindo que técnica seja aplicada a qualquer módulo de uma plataforma.

Com o intuito de aprimorar a utilização da metodologia, propusemos utilizar a base de reflexão computacional desenvolvida para extrair da plataforma os metadados necessários para uma análise de fluxo de dados na plataforma. Aliando a metodologia de reflexão à sobrecarga de tipos, pudemos aplicar técnicas como a análise de alcance de definições, usual no domínio de compiladores, às descrições de *hardware*. Na metodologia proposta, passamos a visualizar a plataforma como um modelo comportamental, possibilitando a aplicação de que algumas técnicas usuais e comprovadamente funcionais no domínio de compiladores, como a geração dos conjuntos *Generate* e *Kill*. Mostramos dois exemplos de como usamos nossa proposta para detectar um erro de sincronismo difícil de ser encontrado por outra técnica e também como podemos extrair paralelismo de uma plataforma usando somente a análise de fluxo de dados.

5.1 Trabalhos Futuros

O desenvolvimento da metodologia de reflexão está em um nível de maturidade elevado, tendo sido publicada em [2] e apresentada em diversas ocasiões. O trabalho mostrado em [22] é um exemplo da aplicação da técnica para introspecção de plataformas. A evolução desta metodologia se restringe à expansão e automação da reflexão para incluir tipos de dados ainda não suportados (ex.: tipos dinâmicos, como listas) ou gerar automaticamente os dicionários, tarefa que, ainda que necessite ser realizada somente uma vez por IP, é realizada manualmente.

No cenário do *Signal Replay* gostaríamos de utilizar como ponto de partida não a execução padrão da plataforma já montada mas sim o *software* base. Hoje, monitoramos na execução controlada os pontos de entrada e saída dos módulos. Em uma execução em *software*, isso corresponde aos parâmetros da função chamada e seus valores de retorno. No nosso exemplo, partiríamos de uma execução puramente em *software* do algoritmo MP3, monitorando as chamadas de função para o procedimento que executa a IDCT, por exemplo, gravando no arquivo os parâmetros passados para a função e o seu retorno. Estes dados podem ser transformados para que os parâmetros sejam injetados em um módulo SystemC e, consequentemente, sua saída seja comparada com os valores de retorno da função. Esta expansão pode eliminar um passo e pode adiantar ainda mais a introdução da verificação no fluxo de projeto. Para *software* IPs, isto significaria que teríamos um IP em teste praticamente imediatamente na primeira fase de projeto, possibilitando a adoção da metodologia mesmo antes da viabilização do projeto.

Já na PDFA, as possibilidades são mais amplas. Além da análise de definições alcançantes, podemos ainda trazer do domínio de compiladores muitas outras técnicas já desenvolvidas e maduras. Há interesse especial em técnicas de geração de código e otimizações. Na geração de código, podemos aproveitar o modelo comportamental do IP para gerar código sintetizável usando as mesmas técnicas de compiladores. Já nas otimizações, as opções são maiores, devido ao grande número de técnicas de otimização adaptáveis, já consolidada pelo estudo de compiladores. Pode-se por exemplo usar as otimizações sobre a descrição no domínio do *software* (ESL), refletindo diretamente no final do projeto, no silício. O objetivo final é alertar o projetista já no início do desenvolvimento que o modelo que ele está trabalhando pode ser otimizado para ocupar uma área menor no silício ou consumir menos, caso seja feito de outra forma. Além disso, técnicas como a eliminação de código morto podem beneficiar muito o campo da ESL, facilitando a transformação do *software* em *hardware* e tornando esta tarefa um procedimento mais direto. A proposta inicial era aproximar as técnicas de compiladores ao domínio ESL. O domínio de compiladores é extenso e demoraremos algum tempo até experimentarmos em descrições de *hardware* as ferramentas, técnicas e algoritmos disponíveis para compiladores.

5.2 Lista de Publicações

(a) Bruno Albertini, Sandro Rigo, Guido Araujo, Cristiano Araujo, Edna Barros e Willians Azevedo. **A Computational Reflection Mechanism to Support Platform Debugging in SystemC**. *CODES+ISSS '07: Proceedings of the 5th IEEE ACM International Conference on Hardware Software Codesign and System Synthesis*, pages 81–86, New York, NY, USA, 2007. ACM.

(b) Sandro Rigo, Rodolfo Azevedo, and Luiz Santos, editores. *Electronic System Level Design: An Open-Source Approach.* Springer, 2011. Capítulos:

3 *Transaction Level Modeling* por Sandro Rigo, Bruno Albertini e Rodolfo Azevedo

5 *Building Platform Models with SystemC* por Rodolfo Azevedo, Sandro Rigo e Bruno Albertini

7 *Debugging SystemC Platform Models* por Bruno Albertini, Sandro Rigo e Guido Araujo

(c) Gabriel Marcilio, Luiz C. V. Santos, Bruno Albertini e Sandro Rigo. **A Novel Verification Technique to Uncover Out-of-order DUV Behaviors.** In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 448–453, New York, NY, USA, 2009. ACM.

(d) Bruno Albertini, Sandro Rigo, Flavia Santos e Guido Araujo. **Platform Data-Flow Analysis**. Apresentação oral no HPPC'11 (*Workshop on Hardware-support for Parallel Program Correctness*, evento da MICRO-44 (*International Symposium on Microarchitecture*, IEEE).

(e) Bruno Albertini, Sandro Rigo e Guido Araujo. **Computational Reflection and its Application to Platform Verification**. In *Design Automation for Embedded Systems*, pages 1–17, Springer Netherlands, ISSN: 0929-5585, DOI: 10.1007/s10617-011-9082-6.

Apêndice A

Links

Este Apêndice foi adicionado para que o leitor possa ter acesso facilitado aos sítios das ferramentas descritas no texto. A lista de *links* não é exaustiva, mas procuramos colocar as principais referências de onde pode-se começar a explorar determinado assunto.

ArchC

`http://www.archc.org`

O ArchC é o simulador de arquiteturas usado para simular os processadores deste trabalho.

ARM

`http://www.arm.com`

RealView Debugger (parte do RealView Development Suite)

`http://www.arm.com/products/tools/software-tools/rvds/arm-rv-debugger.php`

C++ Aspect Oriented Programming

`http://www.aspectc.org`

Cadence

`http://www.cadence.com`

CATS (Communication Art Technology Systems)

`http://www.zipc.com`

XModelink SystemC Debugger

`http://www.zipc.com/english/product/xmodelink/scdebugger.html`

CoWare

`http://www.coware.com`

dist10 , fornecido pelo ISO MPEG Audio Subgroup Software Simulation Group (1996)
`ftp://ftp.tnt.uni-hannover.de/pub/MPEG/audio/mpeg2/software/technical_report/dist10.tar.gz`
Este algoritmo foi a base para a plataforma MP3, usada como exemplo na Seção 3.4.

Forte Design Systems
`http://www.forteds.com`

GCC_XML por Brad King
`http://www.gccxml.org`

GDB: The GNU Project Debugger
`http://www.gnu.org/software/gdb`

KaSCPar
`http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-sim-tools-kascpar-examples`

Mentor Graphics
`http://www.mentor.com`

Mentor Vista
`http://www.mentor.com/esl/vista/overview`

OpenCores
`http://opencores.org`

Open SystemC Initiative
`http://www.systemc.org`
Repositório de IPs de código aberto em diversos níveis de abstração, incluindo IPs em SystemC.

SPIRIT Consortium
`http://www.spiritconsortium.org`

Synopsys
`http://www.synopsys.com`

Platform Architect
`http://www.synopsys.com/Systems/ArchitectureDesign/Pages/PlatformArchitect.aspx`

SystemVerilog
`http://www.systemverilog.org`

Bibliotecas para Introspecção

AReflection por Adams Arne

http://www.arneadams.com/reflection_doku/index.html

Boost C++ libraries

http://www.boost.org

A Boost não é uma biblioteca de reflexão, mas pode ser usada para introspecção.

CPPReflect por Fábio Lombardelli

http://sourceforge.net/projects/cppreflect/

CPP Reflection Package por Konstantin Knizhnik

http://www.garret.ru/~knizhnik/

C++ python interfacing: pyplusplus

http://www.language-binding.net

Outra biblioteca que não é de reflexão. O pyplusplus é um *wrapper* entre o ambiente de execução C++ e o ambiente interpretado do Python, que possui reflexão nativa. Uma abordagem bastante utilizada por um grupo de pesquisa da Universidade de Milão.

OpenC++ por Shigeru Chiba

http://opencxx.sourceforge.net

Esta é uma biblioteca capaz de extrair meta-objetos de classes C++, que podem posteriormente serem utilizados para a instrospecção.

ROOT Project

http://root.cern.ch

Esta é a biblioteca escolhida para implementar o mecanismo de reflexão descrito no Capítulo 3.

Referências Bibliográficas

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] Bruno Albertini, Sandro Rigo, Guido Araujo, Cristiano Araujo, Edna Barros, and Willians Azevedo. A computational reflection mechanism to support platform debugging in systemC. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 81–86, New York, NY, USA, 2007. ACM.

[3] B. Bailey, G. E. Martin, and A. Piziali. *ESL design and verification: a prescription for electronic system-level methodology*. The Morgan Kaufmann series in systems on silicon. Morgan Kaufmann, 2007.

[4] Giovanni Beltrame, Cristiana Bolchini, Luca Fossati, Antonio Miele, and Donatella Sciuto. ReSP: a non-intrusive transaction-level reflective MPSoC simulation platform for design space exploration. In *ASP-DAC '08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 673–678, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[5] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. *SIGPLAN Not.*, 43:316–326, June 2008.

[6] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[7] David Déharbe and Sergio Medeiros. Aspect-oriented design in systemC: implementation and applications. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 119–124, New York, NY, USA, 2006. ACM Press.

[8] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. Introspection in system-level language frameworks: Meta-level vs. integrated. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10382–, Washington, DC, USA, 2003. IEEE Computer Society.

[9] W. Ecker, V. Esen, and M. Hull. Implementation of a transaction level assertion framework in systemC. *Design, Automation and Test in Europe Conference and Exhibition*, 0:167, 2007.

[10] J. Ferber. Computational reflection in class based object-oriented languages. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 317–326, New York, NY, USA, 1989. ACM Press.

[11] Christian Genz and Rolf Drechsler. Overcoming limitations of the systemC data introspection. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 590–593, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[12] Frank Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[13] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *Workload Characterization, Annual IEEE International Workshop*, 0:3–14, 2001.

[15] A. Habibi and S. Tahar. Towards an efficient assertion based verification of systemC designs. *High-Level Design, Validation, and Test Workshop, IEEE International*, 0:19–22, 2004.

[16] Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. Parmibench - an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, 9:45–48, 2010.

[17] Atsushi Kasuya and Tesh Tesfaye. Verification methodologies in a TLM-to-RTL design flow. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 199–204, New York, NY, USA, 2007. ACM.

[18] Dohyung Kim, Youngmin Yi, and Soonhoi Ha. Trace-driven HW/SW cosimulation using virtual synchronization technique. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 345–348, New York, NY, USA, 2005. ACM.

[19] James Lapalme, El Mostapha Aboulhamid, and Gabriela Nicolescu. A new efficient EDA tool design methodology. *Trans. on Embedded Computing Sys.*, 5(2):408–430, 2006.

[20] Jing-Wun Lin, Chen-Chieh Wang, Chin-Yao Chang, Chung-Ho Chen, Kuen-Jong Lee, Yuan-Hua Chu, Jen-Chieh Yeh, and Ying-Chuan Hsiao. Full system simulation and verification framework. *Information Assurance and Security, International Symposium on*, 1:165–168, 2009.

[21] Peter Lisherness and Kwang-Ting (Tim) Cheng. Scemit: a systemc error and mutation injection tool. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 228–233, New York, NY, USA, 2010. ACM.

[22] Gabriel Marcilio, Luiz C. V. Santos, Bruno Albertini, and Sandro Rigo. A novel verification technique to uncover out-of-order DUV behaviors. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 448–453, New York, NY, USA, 2009. ACM.

[23] Deepak A. Mathaikutty and Sandeep K. Shukla. Mining metadata for composability of IPs from systemC IP library. *Design Automation for Embedded Systems*, 12:63–94, June 2008. http://www.springerlink.com/content/y875270qr501h668.

[24] M. Metzger, A. Anane, F. Rousseau, J. Vachon, and E. M. Aboulhamid. Introspection mechanisms for runtime verification in a system-level design environment. *Microelectronics Journal*, 40(7):1124–1134, 2009. Mixed-Technology Testing; Rapid System Prototyping.

[25] Silvio Misera, Heinrich Theodor Vierhaus, and André Sieber. Simulated fault injections and their acceleration in systemC. *Microprocess. Microsyst.*, 32(5-6):270–278, 2008.

[26] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Lussy: An open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 10:73–104, 2005. 10.1007/s10617-006-9044-6.

[27] Sandro Rigo, Rodolfo Azevedo, and Luiz Santos, editors. *Electronic System Level Design: An Open-Source Approach*. Springer, 2011.

[28] Frank Rogin, Christian Genz, Rolf Drechsler, and Steffen Rülke. An integrated systemC debugging environment. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, volume 10 of *Lecture Notes in Electrical Engineering*, pages 59–71. Springer Netherlands, 2008.

[29] S. Roiser and P. Mato. The SEAL C++ reflection system. In *CHEP '04: Presented in the Computing in High Energy and Nuclear Physics congress (CHEP'04)*, Interlaken, Switzerland, September 2004. CERN, CERN.