Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Análise de Cobertura Funcional na Fase de Integração de Blocos de Circuitos Digitais

Cássio Leonardo Rodrigues

Tese de Doutorado submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I, como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências no domínio da Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Jorge César Abrantes de Figueiredo
Dalton Dario Serey Guerrero
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

R696a Rodrigues, Cássio Leonardo.
Análise de cobertura funcional na fase de integração de blocos de circuitos digitais / Cássio Leonardo Rodrigues. — Campina Grande, 2010.
172 f.: il.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientadores: Prof. Dr. Jorge César Abrantes de Figueiredo, Prof. Dr. Dalton Dario Serey Guerrero.

1. Verificação e Validação de Dados. 2. Verificação. 3. Validação e Análise de Cobertura Funcional. I. Título.

CDU – 004.415.5(043)

NÁLISE DE COBERTURA FUNCIONAL NA FASE DE INTEGRAÇÃO DE BLOCOS DE CIRCUITOS DIGITAIS"

CASSIO LEONARDO RODRIGUES

TESE APROVADA EM 14.04.2010

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador(a)

DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)

JOSEANA MACÊDO FECHINE, D.Sc
Examinador(a)

ELMAR UWE KURT MELCHER, Dr.
Examinador(a)

ALISSON VASCONCELOS DE BRITO, D.Sc
Examinador(a)

EDNA NATIVIDADE DA SILVA BARROS, Drª
Examinador(a)

JOSÉ ANTÔNIO GOMES DE LIMA, D.Sc
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Um dos maiores desafios no projeto de um circuito digital é assegurar que o produto final respeita suas especificações. A verificação funcional é uma técnica amplamente empregada para certificar que o projeto do circuito digital respeita suas especificações. Devido à complexidade dos circuitos digitais, os engenheiros criam projetos hierárquicos, decompondo blocos complexos em blocos mais simples. Conseqüentemente, a verificação funcional é realizada de acordo com a decomposição hierárquica do projeto. No entanto, a fase de composição não é devidamente tratada pelas metodologias de verificação funcional. Elas não determinam como proceder de maneira sistemática para se reduzir o tempo de integração e explorar novos cenários que podem surgir da interação entre blocos. Este trabalho apresenta uma abordagem de verificação funcional específica para a fase de composição de blocos de projeto. Esta abordagem é capaz de promover o reuso de componentes de verificação, a preservação de critérios de cobertura dos blocos, a exploração de novos cenários emergentes da interação entre blocos e redução do tempo na verificação funcional. Os experimentos realizados neste trabalho proporcionaram melhoramentos significativos em projetos de circuitos digitais que foram desenvolvidos no âmbito acadêmico. Por meio de métricas de cobertura estrutural, foi mostrado que as novas especificações de cobertura funcional podem exercitar trechos de código que não tinham sido exercitados até o momento da integração.

Abstract

One of the biggest challenges in a digital circuit design is to assure that the final product complies with its specifications. Functional verification is a widely employed technique to certify that the digital circuit design complies with its specifications. Due to complexity of digital circuits, the engineers create hierarchical designs, breaking a complex block into simpler blocks. Hence, the functional verification is performed in accordance with the hierarchical decomposition for the design. However, the composition phase is not well treated by the functional verification methodologies. They do not determine how to proceed in a systematic way to reduce integration time and explore new scenarios that may arise from the interaction between blocks. This work presents a functional verification approach that is specific for the design blocks composition phase. This approach is able to promote the reuse of verification components, the preservation of the coverage criteria of the blocks, the exploitation of new scenarios emerging from the interaction of blocks and time reduction in functional verification. The experiments in this work provided significant improvements in digital circuit designs that were developed in the academic domain. By means of structural coverage metrics, it was shown that the new specification of functional coverage can exercise pieces of code that had not been exercised up to the time of integration.

Dedicatória

Dedico este trabalho aos meus pais (*in memoriam*), a minha irmã Haydée (*in memoriam*), as minhas irmãs, aos meu irmãos, a minha esposa e ao meu filho.

Agradecimentos

Desejo externar meus agradecimentos à Karina Rocha Gomes da Silva e ao Prof. Elmar Melcher pelo apoio técnico para realização deste trabalho. Preciso agradecer também a todos os (ex-)alunos do Laboratório de Arquiteturas Dedicadas da Universidade Federal de Campina Grande, em especial, Henrique, Jorgeluis, Maria de Lourdes, Victor, Ladislau, George, Isaac e Sérgio.

Agradeço aos meus orientadores pelo voto de confiança e oportunidade de trabalhar no Grupo de Métodos Formais (GMF).

Agradeço ao Prof. José Antônio e sua equipe do Brazil-IP da UFPB pela significativa ajuda em parte dos experimentos que conduzi.

Agradeço aos colegas de trabalho da UFCG por fazerem da minha passagem pela Paraíba uma experiência de crescimento pessoal e profissional.

Agradeço à Aninha e Vera pela paciência e compreensão ao longo de toda minha vida acadêmica na UFCG.

Agradeço às agências CAPES e CNPq por financiaram o desenvolvimento do trabalho.

Agradeço a minha família pelo afeto, apoio e paciência incondicionais em toda minha existência.

Conteúdo

Lista de Símbolos

AVM - *Advanced Verification Methodology*

BVM - *Brazil-IP Verification Methodology*

DCT - *Discrete Cosine Transform*

DFT - *Design for Testability*

DPCM - *Differential Pulse Code Modulation*

DUV - *Design Under Verification*

IS - *Inverse Scan*

MPEG - *Moving Picture Experts Group*

MR - *Modelo de Referência*

OVM - *Open Verification Methodology*

RM - *Reference Model*

RTL - *Register Tranfer Level*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O contexto geral deste trabalho é o desenvolvimento de sistemas em uma única pastilha (SoC - *System-on-Chip*). SoC é um circuito integrado que engloba componentes díspares, tais como microprocessador, blocos de memória e interfaces externas, necessários para se realizar alguma computação [88]. O projeto de um SoC é baseado em unidades lógicas reusáveis, previamente validadas, denominadas núcleos de propriedade intelectual (em inglês, *Intellectual Property (IP) cores*). Um *IP core* pode ser classificado como *soft* ou *hard* segundo a sua capacidade de ser editado para se adequar às necessidades do projeto. Este trabalho lida com *soft* IP, descrito em linguagens de programação que operam no nível de transferência entre registradores (*Register Transfer Level* (RTL)). De fato, uma fase específica do desenvolvimento de um *soft IP* está sendo abordada, a fase de composição hierárquica. Nesta fase, blocos de projeto mais simples são agrupados para compor o circuito digital.

A verificação funcional na fase de integração é importante porque grande parte dos comportamentos não esperados de certos sistemas ocorre em função de interações complexas entre os diversos blocos que os compõem [61; 60; 58; 48; 29]. Segundo engenheiros da Cadence [26], a abordagem de verificação funcional na fase de integração *não* pode ser uma simples extensão da abordagem utilizada na verificação dos blocos individualmente. Isto porque à medida que os blocos são combinados, o número de combinações de possíveis interações entre eles cresce exponencialmente. Além disso, o desempenho na execução das simulações decai consideravelmente com o aumento da complexidade dos blocos de projeto.

1.1 Motivação e Contexto

Um dos maiores desafios no projeto de um circuito digital é assegurar que o produto final respeita as suas especificações. Como as funcionalidades providas pelos circuitos digitais têm aumentado ao longo do tempo, a academia e a indústria têm concentrado esforços significativos para que as técnicas de verificação evoluam de modo a garantir produtos de qualidade, em um prazo que não comprometa a competitividade do produto. Uma técnica amplamente empregada para se certificar que o circuito digital respeita suas especificações é a verificação funcional [18; 88; 58; 65; 19; 54; 32; 24; 68].

A verificação funcional envolve quatro componentes básicos: as especificações que o sistema deve obedecer, o projeto do sistema, os mecanismos de simulação para atestar que o projeto está de acordo com as especificações e, por último, um mecanismo para se estimar o nível de confiança na atividade de verificação. Um exemplo deste último componente é a análise de cobertura funcional. Na análise de cobertura funcional, o engenheiro especifica um conjunto de metas que devem ser atingidas durante a verificação funcional. Uma vez que estas metas são atingidas, o processo de verificação funcional pode ser encerrado e uma nova etapa na construção do sistema pode ser iniciada. Assim, a verificação funcional não garante que o projeto está isento de erros, ela garante que o nível de confiança esperado na verificação foi alcançado.

Diante da grande complexidade que um circuito digital possui, se torna obrigatório que o projeto do mesmo seja definido segundo uma decomposição hierárquica para dividir um problema complexo em vários blocos de projeto de menor complexidade. As metodologias de verificação funcional tiram proveito desta mesma decomposição para se obter controlabilidade e observabilidade durante a verificação [18; 54; 31]. A controlabilidade indica quão fácil é, para o engenheiro de verificação, a criação de cenários para a detecção de erros. Observabilidade denota o quão fácil é identificar quando o sistema se comporta como esperado ou não. Blocos de mais baixo nível na decomposição hierárquica apresentam maior controlabilidade e observabilidade [88].

Não existe uma metodologia de verificação que seja consenso entre os diversos fabricantes de circuitos digitais. Cada indústria define as atividades e ferramentas de verifi-

cação levando-se em consideração os requisitos do circuito a ser desenvolvido e os recursos humanos, ferramentais e financeiros disponíveis para o projeto. Apesar desta falta de consenso, o que se observa, em grande parte das metodologias, é que a decomposição hierárquica do projeto faz com que a verificação funcional possua dois conjuntos de especificações de critério de cobertura. Existe um conjunto de especificações para verificar cada bloco, de agora em diante chamada de especificação de bloco, e outro para verificar o sistema resultante da integração dos blocos, chamada de especificação do sistema. Para reduzir o custo da verificação funcional, as metodologias, em geral, enfatizam o reuso de componentes de verificação dos blocos na verificação do sistema [54; 88]. Assim, componentes de geração de estímulos, componentes de monitoração e comparação, entre outros, são utilizados na verificação do sistema da mesma maneira como foram utilizados na verificação dos blocos. Na Figura 1.1, adaptada do trabalho de Wile *et al* [88], está ilustrado um exemplo deste tipo de reuso. No lado esquerdo desta figura, os blocos A e B, com os seus respectivos componentes de geração de estímulos e comparadores, são ilustrados. No lado direito tem-se o sistema integrado, composto por dois blocos A provendo resultados para um bloco B. Nesta integração, o componente de geração de estímulos, originalmente usado para verificação de A, é reusado na integração. Da mesma maneira, o componente comparador da verificação funcional de B é reusado na integração.

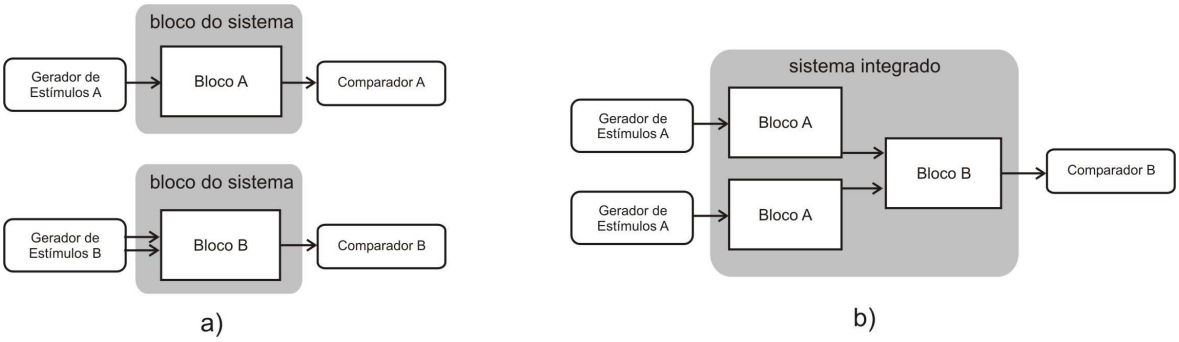

Figura 1.1: Exemplo de integração de blocos com reuso dos componentes de verificação.

Embora a abordagem de verificação funcional alinhada com a decomposição hierárquica do projeto seja capaz de prover reuso de alguns componentes de verificação, a forma como as metodologias tratam a integração dos blocos apresenta problemas. Assumir que os blocos de projeto foram satisfatoriamente verificados do ponto de vista

funcional e enfatizar somente aspectos de comunicação entre blocos, tais como interface e protocolo, podem trazer problemas no desenvolvimento do sistema. Os blocos de projetos são parametrizados para serem reusados em diversos sistemas. Se por um lado esta parametrização viabiliza a configuração do bloco de acordo com a aplicação, por outro ela representa um custo adicional para verificação funcional deste bloco [43; 52]. As diversas possibilidades de parametrização do bloco tornam a especificação de cobertura funcional uma atividade complexa, permitindo que certos cenários importantes não sejam considerados na verificação funcional.

De fato, a verificação funcional na fase de integração não deve ser dirigida somente pelo reuso dos componentes do ambiente de verificação. Ela deve ser dirigida também à exploração de novos cenários para a detecção de erros que podem surgir da interação entre os blocos integrados. Por exemplo, na Figura 1.1, os critérios de cobertura que existiam no bloco comparador de A deixam de existir na integração. Assim, a verificação funcional da integração pode ser concluída sem que as funcionalidades especificadas e verificadas no bloco A em separado sejam propagadas para o bloco B durante a verificação da integração. Os gráficos da Figura 1.2 ilustram uma situação real em que ocorre a redução da cobertura na verificação funcional da integração. No gráfico da parte superior, estão registradas as linhas que foram exercitadas na verificação funcional isolada de um bloco que compõe o decodificador de vídeo MPEG 4 [69]. No gráfico da parte inferior, estão registradas as linhas de código que foram exercitadas quando o referido bloco está integrado com um bloco vizinho. O número maior de barras brancas na parte inferior significa que o número de linhas cobertas na verificação da integração é o menor que o número linhas cobertas na verificação do bloco em separado.

Diversos trabalhos têm se dedicado a explorar cenários complexos que podem surgir da interação entre os diversos componentes que compõem o sistema [49; 61; 50; 45; 86; 48]. No entanto, estes trabalhos se baseiam em exploração de espaço de estados. Um espaço de estados é uma estrutura que registra todos estados alcançáveis pelo sistema, bem como os eventos que fazem com que o sistema evolua de um estado para outro. Técnicas desta natureza apresentam o problema conhecido como explosão do espaço de estados [84], caracterizado pelo fato de que o espaço de estados de um sistema pode ser demasiado grande para ser armazenado na memória de um computador.

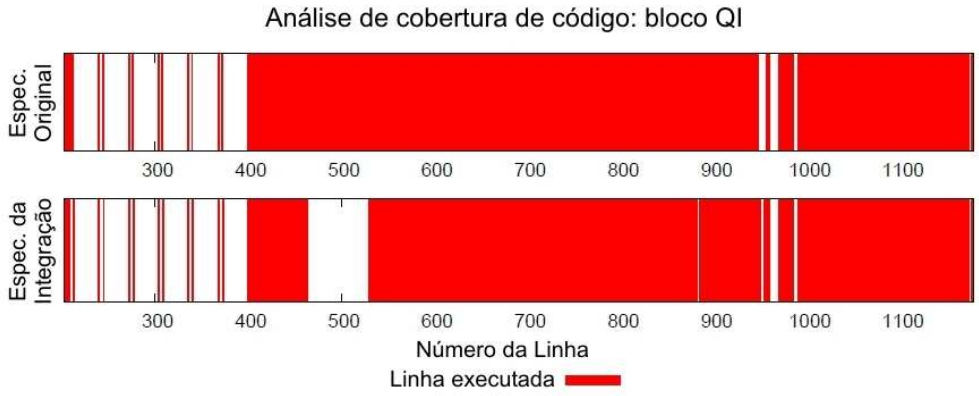

Figura 1.2: Análise de cobertura de linha de código de um bloco do decodificador de vídeo MPEG 4.

O reuso dos componentes de verificação também pode levar à caracterização de critérios de cobertura que não são alcançáveis em um tempo aceitável. Considerando-se um cenário de integração envolvendo dois blocos, B1 e B2, sendo que os resultados produzidos na saída de B1 são usados para alimentar a entrada de B2. Esta integração pode resultar em um cenário conforme ilustrado no gráfico da Figura 1.3. Os blocos B1 e B2 quando verificados individualmente alcançam a cobertura esperada, mas a verificação funcional da integração não atinge 100 % em um tempo aceitável. Como a observabilidade e controlabilidade tendem a diminuir à medida que os blocos de projeto são integrados para formar blocos mais complexos, cenários como esses podem se tornar um impasse para o engenheiro. Isto porque o engenheiro não dispõe de mecanismos para concluir se os critérios de coberturas são de fato inalcançáveis porque o bloco B1 restringe o conjunto de estímulos fornecidos ao B2, ou se o componente gerador de estímulos de B1 não foi configurado de modo a gerar a entrada necessária para satisfazer critérios de B2. Este impasse é bastante caro para o projeto. Ele pode provocar a aprovação de um projeto que não foi devidamente verificado, pois certos critérios de cobertura não foram atingidos, ou então há um desperdício de recursos na tentativa de se atingir critérios de cobertura que não são alcançáveis.

A Figura 1.4 é uma ilustração do diagrama de blocos do decodificador de vídeo MPEG 4 [69]. Na integração dos blocos DCDCT e SI do referido sistema, ocorreu o problema mencionado no parágrafo anterior. A partir da Figura 1.5, é possível perceber que a verificação funcional dos blocos DCDCT e SI alcança 100% em tempo satisfatório. No entanto, quando

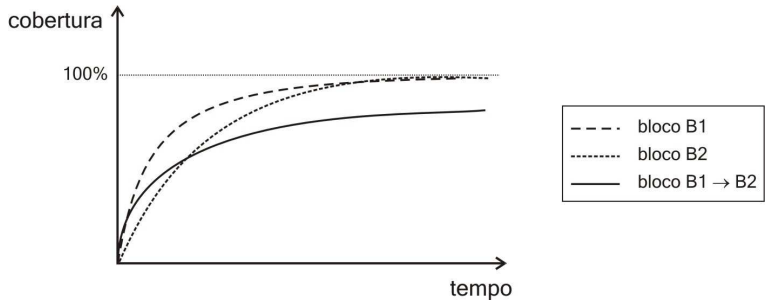

Figura 1.3: Cenário de integração no qual critérios de cobertura deixam de ser alcançáveis.

ocorre a verificação funcional da integração de ambos, a cobertura alcançada deixa de atingir 100%. Isto porque o componente gerador de estímulos do bloco DCDCT não estava configurado para satisfazer os critérios de coberturas situados na saída do bloco SI. Esta constatação, realizada de forma *ad hoc*, consumiu 16 horas de dois engenheiros de verificação.

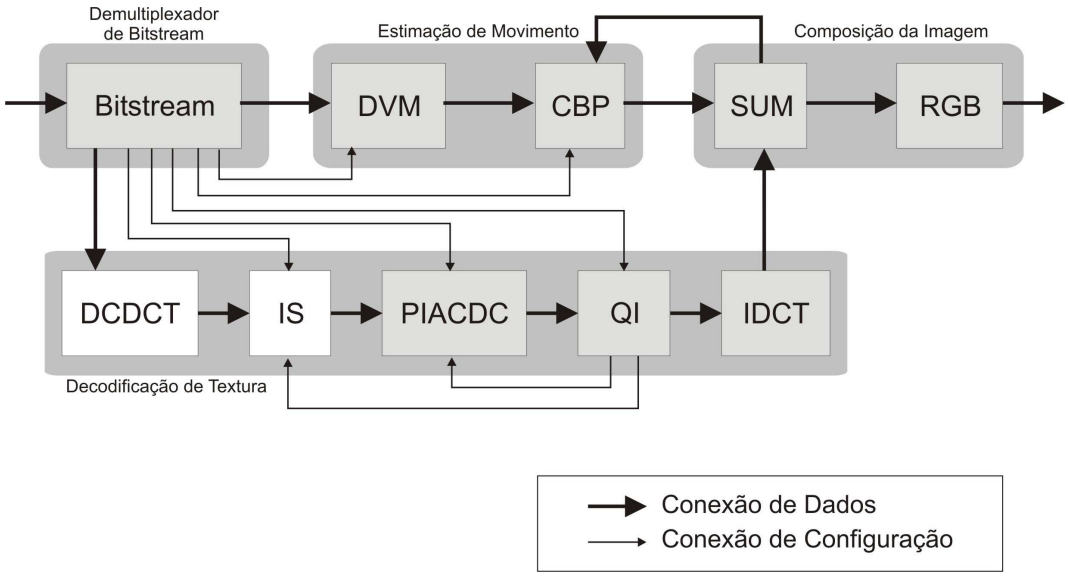

Figura 1.4: Diagrama de blocos do decodificador de vídeo MPEG 4.

1.2 Objetivos

O objetivo deste trabalho é definir um conjunto de procedimentos que devem ser seguidos na fase de integração de blocos de projeto de circuitos digitais para que novos cenários emergentes da interação entre blocos sejam considerados na verificação funcional. Este objetivo

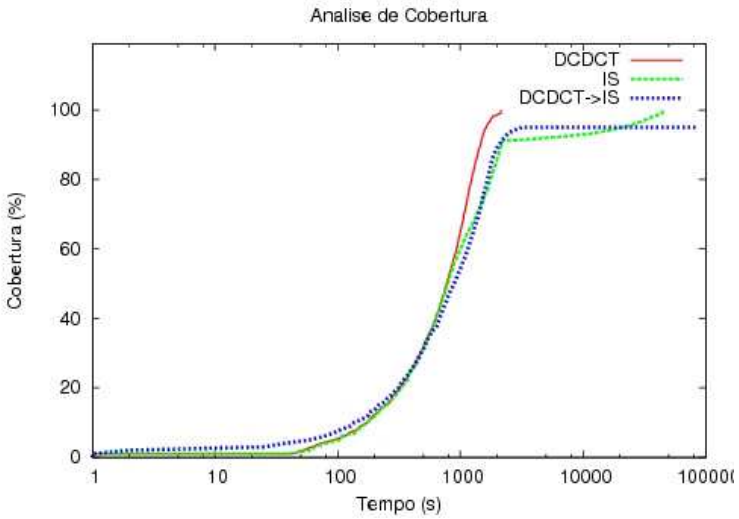

Figura 1.5: Análise de cobertura funcional na integração dos blocos DCDCT e SI do MPEG 4.

maior pode ser decomposto nos seguintes objetivos específicos:

- caracterizar os cenários em que é possível obter novas especificações de cobertura funcional para cada bloco da integração;
- definir algoritmos para obter novas especificações de cobertura funcional para os blocos da integração;
- definir algoritmos para auxiliar o engenheiro de verificação a decidir se a cobertura que é 100 % inalcançável na fase de integração pode ser resolvida ou não com a reconfiguração de geradores de estímulos.

1.3 Metodologia

O desenvolvimento deste trabalho partiu de uma formalização dos principais elementos do ambiente de verificação funcional e que são comuns à grande maioria das metodologias de verificação funcional. Para tal formalização, a teoria dos conjuntos foi utilizada [41; 36; 35]. A partir de tal formalização e da integração de blocos de projeto em sua forma canônica, foram identificados os cenários em que é possível obter novas especificações de cobertura funcional para cada bloco da integração. Em seguida, foi definido como obter as novas

especificações para cada bloco da integração.

Para quantificar o melhoramento decorrente das novas especificações de cobertura funcional obtidas a partir do trabalho proposto, foram realizados experimentos com projetos de sistemas legados, desenvolvidos no contexto do programa Brazil-IP [9]. O melhoramento obtido é quantificado através de métricas de cobertura estrutural: cobertura de linha de código e cobertura de ramificação de código. A cobertura de linha mede o percentual de linhas que foram exercitadas durante a simulação. A cobertura de ramificação de código mede o percentual de estruturas condicionais exercitadas durante a simulação. Métricas de cobertura estrutural são capazes de identificar trechos de código que não foram exercitados durante a verificação funcional. Conseqüentemente, elas são importantes porque comportamentos não esperados do sistema podem ter suas origens justamente nestes trechos de código não exercitados.

Os experimentos com projetos de circuitos digitais desenvolvidos no contexto do programa Brazil-IP também são a base para uma discussão sobre como a ordem de integração de blocos pode influenciar o tempo de execução da verificação funcional. Para realização dos experimentos, foram escolhidos projetos que usam metodologia de verificação funcional diferentes para assim fornecer indícios de que o trabalho proposto não está restrito a uma única metodologia de verificação funcional. A generalidade da técnica proposta também é discutida de forma discursiva por meio de revisão bibliográfica.

1.4 Contribuições

A principal contribuição deste trabalho é uma forma de melhorar a qualidade da verificação funcional, fazendo com que ela seja capaz de cobrir cenários complexos que não foram previstos pelo engenheiro de verificação. De fato, muitos outros trabalhos têm se dedicado a tal melhoramento [49; 61; 50; 45; 86; 48; 16]. Estes trabalhos, porém o fazem através de técnicas baseadas em exploração de espaço de estados. Técnicas desta natureza apresentam o problema conhecido como explosão de espaço de estados [84], caracterizado pelo fato de que o espaço de estados de um sistema pode ser demasiado grande para ser armazenado na memória de um computador. Neste trabalho, o melhoramento da verificação funcional é baseado na teoria dos conjuntos, sem a necessidade de se construir espaço de estados e con-

seqüentemente sem a necessidade de se empregar técnicas custosas para aliviar o problema de espaço de estados [27].

A contribuição principal deste trabalho é decorrente das seguintes contribuições: i) definição de uma estratégia para caracterização e síntese de novos critérios de cobertura funcional de blocos projetos em função de interação de blocos vizinhos e ii) definição de uma estratégia para que os novos critérios de cobertura sejam incorporados de fato ao ambiente de verificação do bloco de projeto, permitindo que suas especificações melhorem cada vez que ele for integrado em uma aplicação.

Os experimentos realizados neste trabalho promoveram também um conjunto significativo de melhoramentos em projetos desenvolvidos no Brazil-IP. A partir de métricas de cobertura estrutural, foi mostrado que as novas especificações de cobertura funcional podem levar ao exercício de trechos que até o momento da integração não tinham sido exercitados. Além disso, em um experimento envolvendo o projeto do decodificador de vídeo MPEG 4, as novas especificações obtidas pelo trabalho proposto fizeram com que um erro fosse apontado pelo ambiente de verificação funcional. Isto não significa, porém, que o projeto não está de acordo com as suas especificações. O referido projeto foi simulado tendo como modelo de referência o código XVid [2], isto é, um código produzido por terceiros. Assim, este erro pode ter sido apontado em função do erro na implementação do XVid.

1.5 Estrutura do Documento

O restante deste documento está estruturado em cinco capítulos. Nas seções seguintes, é apresentado um resumo de cada um deles.

1.5.1 Capítulo 2: Conceitos de Verificação Funcional

No Capítulo 2, são apresentados os conceitos básicos do desenvolvimento de circuitos digitais enfatizando a etapa de verificação funcional. As metodologias AVM (*Advanced Verification Methodology*) [42], VMM (*Verification Methodology Manual*) [19], OVM (*Open Verification Methodology*) [10], VeriSC [31] e BVM (*Brazil-IP Verification Methodology*) são apresentadas de forma resumida. A partir deste levantamento, a escolha de VeriSC como metodologia a ser utilizada no trabalho é justificada. Por último, outros trabalhos que também

visam melhorar a qualidade da verificação funcional de circuitos digitais são apresentados.

1.5.2 Capítulo 3: Verificação Funcional na Fase de Integração de Blocos de Projeto

No Capítulo 3, primeiro é apresentada uma abordagem de integração de blocos de projeto. Algumas metodologias prescrevem a construção e verificação de todos os blocos para depois realizar a integração e verificação como um todo [54; 31]. Alguns autores definem esta estratégia como sendo do tipo *big bang* [53; 22]. Este tipo de integração não é adequada porque quando erros na integração ocorrem, o engenheiro tem poucos elementos para se orientar na localização e correção dos mesmos. Conseqüentemente, o tempo de integração pode aumentar, tornando a verificação mais cara. Para evitar tal problema, uma estratégia de integração que é realizada bloco a bloco até que se obtenha todo o sistema é apresentada. Em seguida, as situações em que este tipo de integração bloco a bloco pode melhorar a especificação de cobertura funcional de cada bloco da integração são identificadas. O melhoramento da especificação funcional é obtido de forma a se tornar de fato critérios de cobertura de cada bloco. Isto é, eles não são obtidos apenas para o momento da integração em que foram obtidos. Eles são preservados para outras situações em que o bloco de projeto for reusado. No mesmo momento em que as novas especificações são calculadas, também é analisado se a especificação de cobertura original continuará sendo alcançada na verificação funcional da integração. Para ilustrar a aplicação da abordagem proposta, um exemplo de integração é apresentado. Realizar todas essas operações manualmente pode ser uma atividade tediosa e sujeita a erros. Por esta razão, a solução proposta é discutida do ponto de vista ferramental, apresentando uma arquitetura para uma ferramenta que automatize o processo.

1.5.3 Capítulo 4: Resultados Experimentais

O Capítulo 4 é dedicado a apresentação dos resultados obtidos pela aplicação do trabalho proposto em projetos do programa Brazil-IP [9]. Foram utilizados tanto sistemas legados quanto sistemas ainda em desenvolvimento. O melhoramento obtido é quantificado por meio de análise de mutantes e duas métricas de cobertura estrutural: cobertura de linha de código e cobertura de ramificação de código. A cobertura de linha mede o percentual de linhas que

foram exercitadas durante a simulação. A cobertura de ramificação de código mede o percentual de estruturas condicionais exercitadas durante a simulação. Métricas de cobertura estrutural são capazes de identificar trechos de código que não foram exercitados durante a verificação funcional. Conseqüentemente, elas são importantes porque comportamentos não esperados do sistema podem ter suas origens justamente nestes trechos de código não exercitados. Situações que necessitam de julgamento do engenheiro de verificação para decidir se a cobertura funcional na verificação da integração pode alcançar 100% ou não foram avaliadas através de experimentos com engenheiros que fazem ou que fizeram parte do programa Brazil-IP.

1.6 Capítulo 5: Conclusões

O quinto capítulo é dedicado à conclusão do trabalho. Baseando-se nos resultados obtidos nos capítulos anteriores, pontos importantes tais como generalidade do trabalho proposto e desdobramentos que podem ser abordados em trabalhos futuros são discutidos.

Capítulo 2

Fundamentação Teórica e Trabalhos Relacionados

Neste Capítulo, os conceitos básicos do desenvolvimento de circuitos digitais, enfatizando a etapa de verificação funcional, são apresentados. As metodologias AVM (*Advanced Verification Methodology*) [42], VMM (*Verification Methodology Manual*) [19], OVM (*Open Verification Methodology*) [10], VeriSC [31], IVM (*Interoperable Verificatin Methodology*) [66] e BVM (*Brazil-IP Verification Methodology*) são apresentadas de forma resumida. A partir de informações sobre tais metodologias, a escolha de VeriSC e BVM como metodologias a serem utilizadas no trabalho é justificada. Por último, outros trabalhos que também visam melhorar a qualidade do projeto de circuitos digitais são apresentados.

2.1 O Desenvolvimento de um Circuito Digital

A Figura 2.1, extraída do trabalho de Wile *et al* [88], é uma ilustração das etapas do desenvolvimento de um chip. O desenvolvimento se inicia com o levantamento dos requisitos do sistema. Estes requisitos são obtidos principalmente por meio de pesquisas de mercado e dizem respeito a funcionalidades, tamanho do chip, consumo de energia, velocidade de processamento etc. Estes requisitos se tornam uma especificação de fato na etapa de especificação do hardware. A especificação, geralmente, é feita em um alto nível de abstração, sem tratar de detalhes de implementação. Com a especificação do sistema em mãos, o engenheiro define a arquitetura do sistema, detalhando os seus principais componentes internos.

Na etapa seguinte, o engenheiro define um projeto por meio de abstrações ainda de alto nível. Este projeto é um passo intermediário para se obter a implementação em RTL do sistema. Este modelo em RTL pode ser produzido em sua totalidade pelo engenheiro, mas comumente ele é montado fazendo-se reuso de componentes de outros projetos ou de outros fabricantes.

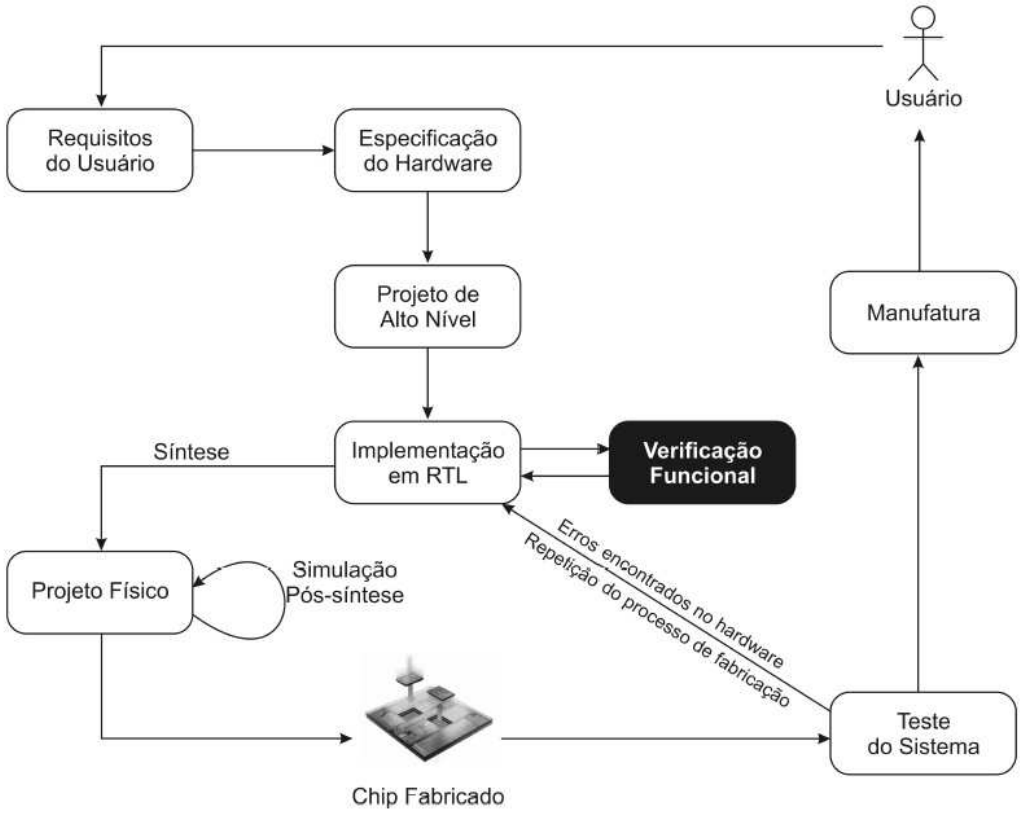

Figura 2.1: Etapas para o desenvolvimento de hardware.

Na etapa de verificação funcional, as informações produzidas até então são utilizadas para se atestar que o modelo RTL de fato implementa as funcionalidades do sistema. Depois que o modelo RTL é aprovado na etapa de verificação funcional, o projeto físico do sistema é obtido automaticamente por meio de ferramentas de síntese. Este projeto físico é uma *netlist* em nível de portas lógicas. Assim que a síntese é realizada, o projeto físico é validado por meio de simulação pós-síntese. A partir deste momento, aspectos físicos do dispositivo, como por exemplo, atrasos das portas lógicas, passam a ser considerados. A simulação pós-síntese é essencial para determinar se os requisitos de tempo estão sendo respeitados e se pode ser alcançado um desempenho melhor do dispositivo por meio de circuitos mais otimizados. Mesmo depois de fabricado, o dispositivo passa por testes com aplicações do usuário

final. Freqüentemente, erros que escaparam nas etapas anteriores de verificação são detectados na fase de testes do sistema. Quando isto acontece, o processo de fabricação precisa ser repetido. Quando o dispositivo se comporta conforme esperado, ele é manufaturado e comercializado.

2.2 Conceitos de Verificação Funcional

O ambiente de verificação (ou *testbench*, em inglês) modela o universo no qual o projeto sendo verificado, de agora em diante designado por DUV (*Design Under Verification*), vai funcionar depois de concluído. Ele contém o código que é responsável por gerar, observar e verificar dados de entrada/saída do DUV. Em geral, este código é decomposto em componentes geradores de estímulos, componentes de monitoração, modelos de referência, comparadores e, em alguns casos, marcadores (*scoreboards*). Estes componentes podem ser implementados em linguagens de descrição de hardware, tais como SystemC ou SystemVerilog, ou em linguagens de propósito geral, tais como C/C++ ou Java. Nas subseções seguintes, é apresentada uma visão geral dos componentes do ambiente de verificação funcional.

2.2.1 Geradores de Estímulos

O componente gerador de estímulos é responsável por produzir as entradas que são utilizadas para a execução do DUV. Assim, este componente deve modelar o comportamento do usuário ou dos demais sistemas que fornecem algum dado para a operação do DUV. Diante da realidade de que um projeto pode ser desenvolvido uma única vez e ser reusado diversas vezes em diferentes contextos, o gerador não deve se restringir a valores que podem ser fornecidos pelos vizinhos do DUV no sistema corrente. De fato, o foco da geração de estímulos deve ser o conjunto de valores que o DUV é capaz de operar. Ao proceder desta maneira, o engenheiro tem mais chances de exercitar cenários mais complexos e que ocorrem raramente (em inglês, estes cenários são chamados de *corner cases*).

Os estímulos devem ser gerados de forma que seja possível repetir uma determinada simulação quantas vezes forem necessárias pelo engenheiro. Isto permite, por exemplo, que o engenheiro detecte um erro, faça as devidas alterações para removê-lo e em seguida repita a simulação para se certificar de que tal o erro foi removido de fato e nenhum outro erro foi in-

serido pelas modificações realizadas no projeto. Em geral, duas abordagens são empregadas para a geração de entradas do projeto: geração aleatória e geração direta. Na abordagem aleatória, o engenheiro faz uso de funções que geram seqüências pseudo-aleatórias de números, isto é, seqüências que podem ser previstas conhecendo-se o número inicial. Na abordagem direta, cada estímulo é individualmente definido pelo engenheiro. Esta abordagem é importante nos momentos iniciais do projeto, quando ele ainda possui erros crassos. A abordagem aleatória tem o potencial de exercitar mais funcionalidades do DUV que a abordagem direta, pois as seqüências de números pseudo-aleatórias podem exercitar o DUV de maneiras não previstas pelo engenheiro.

2.2.2 Monitores

O monitor é a parte do ambiente de verificação que é responsável por observar as entradas e saídas do DUV. A partir desta observação, o engenheiro pode realizar análise de cobertura funcional. Este tipo de análise, que será apresentado nas seções seguintes, permite avaliar a qualidade da verificação segundo as funcionalidades que o sistema deve implementar. Um monitor também pode conter uma lógica para que os valores observados por ele sejam convertidos para níveis de abstrações diferentes, conforme as necessidades do projeto. Por exemplo, ele pode converter um valor observado na saída do DUV, considerado de baixo nível, para um nível de abstração mais alto para facilitar a análise pelo engenheiro.

Em ambientes de verificação sofisticados, à medida que a verificação funcional é executada, as informações coletadas pelos monitores podem ser exploradas para se re-configurar os componentes geradores de estímulos. Esta re-configuração tem como objetivo simular uma quantidade maior de funcionalidades. Este tipo de abordagem é chamado de geração dirigida por cobertura [37; 46].

2.2.3 Modelos de Referência

Ao fornecer uma entrada para o DUV, o engenheiro precisa de algum mecanismo para determinar se o resultado produzido na saída é de fato o resultado esperado. Uma forma de determinar este valor esperado é por meio de uma re-implementação do DUV em uma linguagem de mais alto nível que a linguagem usada para a descrição do DUV. Esta re-implementação

é chamada de modelo de referência (em inglês, *reference model*). O modelo de referência calcula todas as saídas esperadas do sistema baseando-se nas entradas que são utilizadas durante a verificação funcional. Em diversas metodologias, a entidade responsável por atestar que o resultado produzido pelo DUV está certo ou errado é chamada de *scoreboard*.

Idealmente, o DUV e o modelo de referência devem ser implementados por engenheiros diferentes para diminuir a possibilidade de que uma interpretação equivocada da especificação do sistema seja repetida no DUV e no ambiente de verificação. Se esta repetição ocorrer, a verificação funcional pode aprovar um DUV que não se comporta conforme o esperado.

2.2.4 Verificadores

Um verificador (em inglês, *Checker*) é a parte do ambiente de verificação que valida, do ponto de vista funcional, se o projeto está funcionando conforme determinado pela especificação. A quantidade de estímulos usados na verificação funcional é grande o suficiente para tornar impraticável a comparação manual dos resultados produzidos pelo DUV com os resultados esperados de fato. Assim, o verificador compara automaticamente o resultado produzido pelo DUV com o resultado produzido pelo modelo de referência. Se os resultados são iguais, a simulação prossegue normalmente. Em caso contrário, o verificador emite uma mensagem notificando o erro. Em alguns casos, o verificador não existe como uma entidade explícita do ambiente de verificação. De acordo com a necessidade do projeto, a comparação que é realizada pelo verificador é feita pelo *scoreboard*.

2.2.5 Abordagens *Black Box*, *Grey Box* e *White Box*

De acordo com a visibilidade do DUV, existem três abordagens de verificação funcional: *Black Box*, *White Box* e *Grey Box*.

Na abordagem *Black Box*, a verificação funcional é realizada sem nenhum conhecimento da implementação real do DUV. A verificação é acompanhada nas interfaces do DUV, sem acesso direto aos estados internos dele e sem conhecimento de sua estrutura e implementação. Esta abordagem possui a vantagem de não depender de qualquer detalhe de implementação. Mesmo que o DUV seja modificado durante a fase de verificação, o ambiente de verificação não precisa ser alterado se a interface não for mudada. Esta abordagem também

permite que os elementos do ambiente de verificação sejam reutilizados na verificação de outros dispositivos. O ponto negativo dessa abordagem é que se perde um pouco do controle da parte interna da implementação do DUV.

Com a abordagem *White Box*, o engenheiro tem uma visibilidade completa da estrutura interna do DUV. A vantagem de se usar *White Box* é que se pode testar diretamente funções do dispositivo sendo verificado, simplificando o processo de localização dos erros. A desvantagem dessa abordagem é que ela é altamente acoplada a detalhes de implementação do DUV. Conseqüentemente, alterações no DUV podem resultar em modificações no ambiente de verificação. Além disso, é necessário ter conhecimento de detalhes internos para a criação de cenários de testes e saber quais as respostas que devem ser esperadas. Conseqüentemente, o reuso dos elementos do ambiente de verificação em outros dispositivos pode ser mais complicado.

A abordagem *Grey Box* possui características das duas abordagens. Ela procura resolver o problema da falta de controlabilidade do *black Box* e da dependência de implementação do *white Box*. Um exemplo de um caso de teste típico *Grey Box* pode ser usado para aumentar as métricas de cobertura. Nesse caso, os estímulos são projetados para linhas de código específicas ou para verificar funcionalidades específicas.

2.3 Análise de Cobertura

Um aspecto crítico da verificação funcional é a detecção de seu término. Dado que a comparação dos resultados do Modelo de Referência e do DUV é realizada através de simulação, para a grande maioria dos sistemas atuais, não é possível fazer com que todos os cenários possíveis do projeto sejam exercitados em um tempo aceitável. O que se costuma fazer é empregar uma técnica para detectar se o projeto foi suficientemente exercitado durante a verificação funcional. Esta técnica é conhecida como análise de cobertura.

A análise de cobertura é uma técnica usada para medir o progresso da verificação em relação a algum conjunto de critérios. A análise de cobertura pode ser compreendida como sendo um conjunto de metas que devem ser atingidas durante a verificação funcional. Essas metas podem ser especificadas em função de diversos critérios. De acordo com esses critérios, podemos classificar dois tipos de cobertura principais: análise de cobertura de código e

análise de cobertura funcional.

2.3.1 Análise de Cobertura de Código

Para a cobertura de código, a ferramenta de análise de cobertura vai reportar quais partes do código foram executadas e quais não foram durante a simulação. Este tipo de análise é importante por revelar ao engenheiro se existe alguma parte do projeto que não foi exercitada. A existência de partes do projeto que não foram exercitadas é ruim porque elas podem conter algum erro. Este tipo de cobertura requer algum tipo de instrumentação do código. Esta instrumentação consiste de pontos de observação no código para registrar se tal parte foi exercitada de fato. As seguintes métricas podem ser usadas para este tipo de cobertura: linhas de código, caminhos de execução e expressões.

Na cobertura de linhas de código, a ferramenta mede quais linhas exatamente foram executadas e quais não foram. Para se alcançar 100% de cobertura de código é necessário compreender quais condições lógicas devem ser satisfeitas para se alcançar as linhas descobertas. Contudo, é muito comum que a codificação defensiva leve à produção de blocos de comandos que nunca são executados, fazendo com que a cobertura total nunca seja alcançada.

A cobertura de caminhos de execução mede as possíveis seqüências de linha de comando que podem ser executadas em um projeto. A existência de comandos de fluxo de controle, tais como estruturas condicionais do tipo *if then else*, faz com que existam vários caminhos de execução. Por exemplo, temos um caminho em que o bloco *then* é executado e temos outro referente ao bloco *else*. Este tipo de cobertura é bem mais precisa que a cobertura de linhas de código, pois um erro pode ser revelado somente quando uma seqüência específica ocorre. Em contrapartida, o número de seqüências cresce exponencialmente em função do número de comandos de fluxo de controle.

Ainda mais precisa do que a cobertura de caminhos de execução, a cobertura de expressões analisa as diversas instâncias que um caminho de execução pode ocorrer. Por exemplo, a condição de um bloco *if* pode conter uma expressão com o operador lógico *ou*. Naturalmente, esta expressão pode ser satisfeita quando um dos operandos for verdadeiro, fazendo com que existam pelo menos três instâncias do mesmo caminho de execução.

2.3.2 Análise de Cobertura Funcional

A análise de cobertura funcional mede o quanto da especificação original foi exercitado. Ela pode analisar, por exemplo, o nível de ocupação de um *buffer*, a quantidade de pacotes enviados, requisição de barramento etc. Assim, a cobertura funcional está focada no propósito da função implementada enquanto a cobertura de código está focada na execução do código.

A cobertura funcional, portanto, depende do domínio da aplicação. Na prática, isto significa que a especificação dos critérios de cobertura deve ser realizada manualmente pelo engenheiro de verificação, isto é, os critérios de cobertura não podem ser extraídos automaticamente do código em linguagem de descrição do *hardware*. Assim como na análise de cobertura de código, os valores das execuções são extraídos durante a simulação e armazenados em uma base de dados. A partir dessa base, a análise de cobertura ocorre propriamente em função do que foi especificado. Os critérios comumente utilizados para cobertura funcional são os seguintes: cobertura de valores escalares individuais e cobertura cruzada.

Na cobertura de valores escalares, o engenheiro especifica o conjunto de valores relevantes que devem ser observados na verificação, seja como estímulos de entrada, seja como resultados de saída. Exemplos de valores escalares utilizados para este tipo de cobertura são: tamanho de pacote, ocupação de *buffer*, acesso a barramento etc. É uma tarefa complexa especificar e medir cobertura desta natureza, sendo que as medições chegam bem perto de 100% na maioria das vezes [19; 88]. É importante que fique claro que 100% de cobertura não garante que o projeto está isento de erros. Isto que significa que todos os critérios especificados foram totalmente satisfeitos.

A cobertura cruzada de valores[1] mede a ocorrência da combinação de diversos valores. Ela é útil para especificar propriedades do tipo: “um pacote corrompido foi inserido em todas as portas?” ou “todos os *buffers* ficaram preenchidos ao mesmo tempo? ”. A implementação de cobertura cruzada segue o mesmo princípio da cobertura de valores escalares, a diferença é que na cobertura cruzada várias valores são coletados ao mesmo tempo.

[1] *cross coverage*, em inglês

2.4 Níveis de Verificação

Devido à alta complexidade dos circuitos digitais, os engenheiros de projeto usam decomposição hierárquica para dividir um problema complexo em vários outros problemas de menor complexidade. Os engenheiros de verificação aproveitam esta decomposição hierárquica para conduzir a verificação funcional do sistema. De acordo com a decomposição hierárquica do projeto, podem existir vários níveis de verificação funcional. Neste trabalho, vamos considerar os níveis definidos por Vasudevan [85]. O nível mais básico de verificação é o nível de bloco de projeto. Neste nível, um bloco é composto por um simples módulo em RTL ou por um agrupamento de vários módulos menores. Este nível permite uma flexibilidade maior na verificação, sendo que o engenheiro tem como proceder com a abordagem *grey Box*. As coberturas funcionais e de código neste nível são alcançadas em sua totalidade, conseqüentemente, boa parte dos erros é encontrada neste nível de verificação.

Quando alguns blocos de projeto são agrupados segundo alguma afinidade, temos o nível de verificação do subsistema. A partir deste nível, alguns componentes do ambiente de verificação funcional desenvolvidos para a verificação de cada bloco separadamente podem ser reusados, tais como geradores de estímulos, monitores e modelos de referência. No nível de verificação do sistema, todos os subsistemas são agrupados em um só bloco. O foco da verificação neste nível é a execução das funcionalidades do início ao fim para se garantir que cenários que permeiam todos os blocos estão livres de erros.

A partir da verificação do subsistema, o tempo de execução da simulação é bem superior ao tempo de execução do nível anterior. Isto porque a possibilidade de cenários aumenta à medida que os blocos de projeto são integrados. Conseqüentemente, a complexidade de se configurar componentes do ambiente de verificação para explorar cenários específicos também é maior.

2.5 Exemplos de Metodologia de Verificação Funcional

Na Tabela 2.1, é apresentada uma visão geral das seguintes metodologias de verificação funcional:

1. *Advanced Verification Methodology* (AVM) [42];

2. *Verification Methodology Manual* (VMM) [19];

3. *Open Verification Methodology* (OVM) [10];

4. VeriSC [32; 31];

A Tabela 2.1 foi estruturada considerando-se as características básicas que uma metodologia de verificação funcional deve possuir. Segundo Bergeron [18] uma metodologia deve ter quatro características básicas:

- Ser dirigida por coberturas. Devido à grande complexidade dos circuitos digitais, simular todos os cenários possíveis de um projeto é algo impraticável. Assim, é fundamental que o engenheiro tenha a sua disposição mecanismos para especificar quais cenários devem ser simulados de fato. A verificação funcional deve parar somente quando estes cenários forem cobertos. Para este propósito, os engenheiros usam cobertura funcional.

- Suportar a geração aleatória de estímulos. Uma vez que existem critérios de cobertura bem definidos, o engenheiro precisa de suporte ferramental para auxiliar na geração de estímulos que resultem na satisfação destes critérios. Em outras palavras, o engenheiro precisa fornecer as entradas certas para que os objetivos da verificação sejam alcançados. O uso de funções pseudo-aleatórias para geração destas entradas permite que o projeto seja massivamente estressado, fazendo com que cenários mais raros e complexos sejam exercitados. É importante ficar claro que a escolha das funções para geração das entradas depende do conhecimento do engenheiro a respeito do projeto a ser verificado.

- Ser auto-verificável. A quantidade de estímulos usados para atingir critérios de cobertura de um projeto é grande o suficiente para tornar tediosa e suscetível a erros a comparação manual dos resultados produzidos pelo DUV com os resultados esperados de fato. Por esta razão, as metodologias devem dar suporte à determinação do que é de fato um resultado esperado em função da entrada fornecida ao projeto. Além disso, a metodologia deve dar suporte à comparação automática deste resultado com o resultado produzido pelo DUV. Quando for o caso em que os resultados produzidos são diferentes, a metodologia deve reportar ao engenheiro esta discrepância.

- Ser baseada em transações. O DUV é descrito em linguagem RTL, isto é, usando-se um nível de abstração mais baixo, chamado abstração no nível de sinais. Trabalhar o ambiente de verificação todo neste nível de abstração pode ser contraproducente dado que seres humanos não lidam com este nível de abstração na maioria do tempo. Por esta razão, o ambiente de verificação deve operar em um nível de abstração mais alto que o nível de abstração do DUV, chamado de nível de transação. Uma transação é uma operação que se inicia num determinado momento no tempo e termina em outro, sendo caracterizada pelo conjunto de instruções e dados necessários para realizar a operação.

Tabela 2.1: Visão geral das metodologias de verificação funcional.

Metodologia	Suporte a transações	Estímulos aleatórios	Auto-verificável	Dirigida por coberturas
AVM	Sim. A biblioteca `avm_transaction`, implementada a partir de primitivas básicas de SystemVerilog, tais como *put* e *get*, permite que os componentes do ambiente de verificação se comuniquem por meio de transações.	Sim. A biblioteca `avm_random_stiumlus` oferece suporte à geração de seqüências de transações com dados aleatórios. Uma transação com dados aleatórios é definida por meio de variáveis *rand*. Para cada variável *rand* é associada uma função que restringe os possíveis valores de acordo com a necessidade do projeto.	Sim. Existe o conceito de *scoreboard*, que é responsável por re-implementar o DUV no nível de sinais. O *scoreboard* recebe as entradas e saídas do DUV e avalia se os resultados estão corretos.	Sim. A cobertura é especificada através de *covergroup*, *coverpoint* e *bins* de SystemVerilog. O ambiente de verificação é estruturado para finalizar a simulação assim que os critérios de cobertura funcional são satisfeitos.
VMM	Sim. A partir de extensões da classe `vmm_data`, o engenheiro define os dados que são trocados entre os componentes do ambiente de verificação. Os componentes do ambiente de verificação que lidam com as transações de fato são implementados como extensão da classe `vmm_xactor`.	Sim. Através das macros `vmm_atomic_gen` e `vmm_scenario_gen` é possível realizar a geração aleatória de estímulos. Os geradores são projetados para permitir que as funções de restrição sejam externamente configuradas, sem a necessidade de alterações no código fonte. A idéia é permitir que estímulos aleatórios sejam definidos não por meio de geradores completamente novos, mas pela adição de restrições e cenários a geradores que já existem.	Sim. O ambiente de verificação considera o conceito de modelo de referência, que opera no nível de transações, processando as entradas na mesma ordem em que são processadas pelo DUV. O comparador dos resultados que é responsável por esperar até que os resultados produzidos por ambos fiquem prontos.	Sim. A cobertura é implementada a partir dos mecanismos básicos de cobertura funcional de SystemVerilog, tais como `covergroup`, `coverpoint` e `bins`. Além disso, metodologia contém uma série recomendações que o engenheiro deve seguir para se conseguir uma cobertura efetiva sem comprometer o reuso dos componentes de verificação.

Continua na página seguinte.

Metodologia	Suporte a transações	Estímulos aleatórios	Auto-verificável	Dirigida por coberturas
OVM	Sim. Uma transação é um objeto da classe `ovm_transaction`, que encapsula todas as informações necessárias à modelagem da comunicação entre dois componentes. Internamente, a comunicação é implementada da mesma forma como foi definida para a metodologia AVM.	Sim. O engenheiro define as entradas como sendo uma extensão da classe `ovm_sequence_item`. Os atributos dessa classe são definidos usando-se o modificador rand e as funções de restrição que operam sobre estes atributos. Em seguida, o engenheiro define o gerador de estímulos de fato, chamado de *Sequencer*. Este gerador deve ser do tipo `ovm_sequencer`.	Sim. O *scoreboard* é definido pela extensão da classe `ovm_scoreboard`. Assim sendo, o engenheiro pode trabalhar com o *scoreboard* no nível de transação. A metodologia sugere uma seqüência de passos para sua criação e integração no ambiente de verificação.	Sim. A cobertura funcional é implementada em objetos do tipo `ovm_monitor`. Tal como AVM e VMM, a especificação da cobertura é realizada a partir de declarações *covergroup*, *coverpoint* e *bins*.
VeriSC	Sim. O ambiente de verificação é implementado em SystemC, que oferece suporte a transações através da biblioteca SCV. As transações usadas na simulação ficam registradas em arquivos para serem usadas de acordo com as necessidades do projeto. Parte do código para suporte a transações é gerada pela ferramenta que gera o ambiente de verificação.	Sim. Existe o componente gerador de estímulos chamado *Source* que permite a geração de seqüências de valores aleatórios. O gerador faz uso de funções da biblioteca SCV da linguagem SystemC para geração destas seqüências. Os valores são gerados usando-se declarações `scv_bag` e `scv_constraint`.	Sim. O ambiente de verificação trabalha com modelos de referência para determinar os valores que devem ser produzidos pelo DUV. A comparação dos valores produzidos pelo modelo de referência e pelo DUV é realizada por outro componente do ambiente de verificação, chamado *Checker*.	Sim. Existe a biblioteca BVE que dá suporte à cobertura funcional. A cobertura pode ser medida em componentes de monitoração, modelos de referência e *drivers*. Um *driver* é responsável, entre outras coisas, por converter o fluxo de dados no nível de transações para o nível de sinais.

Conforme apresentado na Tabela 2.1, as referidas metodologias satisfazem a todos esses quesitos. Elas se diferem basicamente na forma de implementação dos componentes do ambiente de verificação. AVM, VMM e OVM, por serem implementadas em uma linguagem orientada a objetos, são baseadas em padrões de projetos de software. Um padrão de projeto define solução para um problema recorrente no desenvolvimento de sistemas de software orientados a objetos. Em OVM, por exemplo, a geração do componente de geração de estímulos é baseado no padrão *factory method* [40]. Neste padrão, o projeto fornece uma interface para criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas. Como benefício, ele permite adiar a instanciação para as subclasses, tornando mais flexível a criação de variáveis aleatórias e a configuração de funções de restrição que atuam sobre tais variáveis. Seguindo uma abordagem diferente de implementação, VeriSC tem como ênfase um ambiente de verificação que é parcialmente gerado por meio de ferramenta. O engenheiro provê informações arquiteturais do projeto e a ferramenta gera um molde do ambiente de verificação do projeto. Em seguida, o engenheiro provê as informações que são específicas do projeto em desenvolvimento.

Uma metodologia descendente de VeriSC é a metodologia BVM (Brazil-IP Verification Methodology), implementada em SystemVerilog. BVM preserva fluxo de desenvolvimento da metodologia VeriSC. Como a principal diferença entre ambas é a linguagem de implementação, decidimos considerar VeriSC como sendo a representativa de ambas. Por este motivo, BVM não aparece na Tabela 2.1. Para o desenvolvimento do trabalho, um diferencial importante a favor de VeriSC e BVM é o acesso a projetos legados e em desenvolvimento usando-se estas metodologias. Por causa deste diferencial, vamos considerar VeriSC como metodologia subjacente para o desenvolvimento do trabalho. Na seção seguinte, é apresentada a metodologia VeriSC.

2.6 A Metodologia VeriSC

2.6.1 O Ambiente de Verificação

A metodologia VeriSC propõe um fluxo de implementação em que o ambiente de verificação é construído antes do projeto sob verificação. Este fluxo e a exigência de uma especificação

de interfaces unificada para o ambiente de verificação e para o projeto sob verificação, podem proporcionar economia de até 30% do tempo de verificação funcional [32]. Antes de explicarmos este fluxo, descreveremos o ambiente de verificação com detalhes.

Os elementos que compõem o ambiente de verificação da metodologia VeriSC são aqueles que estão inseridos na área cinza da Figura 2.2. Os componentes existentes são Source, TDriver, TMonitor, Modelo de Referência e Checker. A comunicação entre os componentes, feita usando-se uma estrutura do tipo FIFO (*First In First Out*), está representada por setas mais largas. O ambiente de verificação se comunica com o DUV por interfaces formadas por sinais, ilustradas pelas setas mais finas da figura.

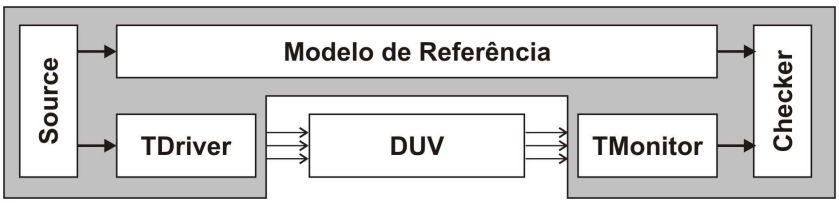

Figura 2.2: Diagrama do ambiente de verificação funcional da metodologia VeriSC.

2.6.2 A Construção do Ambiente de Verificação

A metodologia VeriSC define uma implementação padrão para cada elemento do ambiente de verificação. Esta padronização é o princípio básico para a automação da construção do ambiente de verificação. Uma vez que os elementos são padronizados, é possível identificar quais características são comuns ao ambiente de verificação de qualquer IP e quais características precisam ser especificadas pelo engenheiro de verificação de acordo com a aplicação alvo do IP em desenvolvimento. As partes comuns são mantidas em moldes (*templates*). Estes moldes se tornam um ambiente de verificação de fato quando o engenheiro fornece os detalhes particulares do IP sendo verificado. A combinação do que é comum com o que é específico é realizado pela ferramenta eTBc [64; 33].

O primeiro passo do engenheiro para construir o ambiente de verificação é definir os tipos de comunicação que irão compor o DUV. Para auxiliar na explicação de VeriSC, será considerado o projeto do DPCM (*Differential Pulse-Code Modulation*). O DPCM recebe uma seqüência de amostras na entrada, faz a subtração entre a amostra corrente e amostra

anterior. Caso o resultado da subtração seja um valor fora de uma faixa pré-determinada, o resultado é saturado para um valor dentro da faixa e produzido como resultado de saída. No caso do DPCM, como proposto na Figura 2.3, o sistema é formado por dois blocos: DIF e SAT. O bloco DIF realiza a subtração entre amostras e o bloco SAT verifica se o resultado da subtração está dentro da faixa pré-determinada.

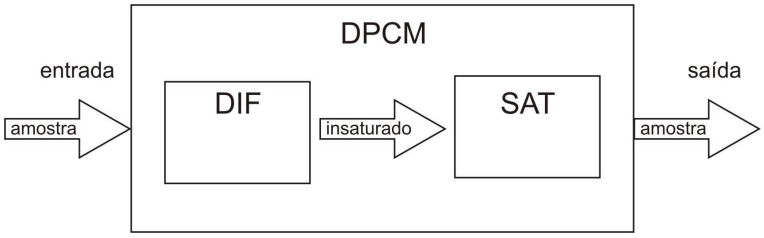

Figura 2.3: Diagrama do DPCM decomposto em dois blocos: DIF e SAT.

O Código 1, nas linhas de 1 a 29, contém a especificação destes tipos (vide [Pessoa, 2007] para informações sobre a gramática da linguagem de especificação de ambientes de verificação da metodologia VeriSC). Existem duas interfaces no bloco DPCM e o tipo de dado para ambas é definido pela estrutura *dpcmSample*. Isto significa que o tipo de dado que entra no DPCM é o mesmo que sai. O tipo de dado que sai do sub-módulo DIF, no entanto, tem tipo diferente, que é um valor insaturado. Um valor insaturado é aquele que precisa ser verificado para saber se está dentro da faixa pré-determinada. Este mesmo tipo serve como entrada do sub-módulo SAT.

Definidos os tipos de comunicação, o engenheiro precisa especificar a hierarquia de blocos do DUV, as interfaces que ele contém e as FIFOs que serão utilizadas para ligar os blocos do projeto conforme apresentado no Código 2. A partir do arquivo formado pelos códigos 1 e 2, o engenheiro vai rodar a ferramenta eTBc, que vai gerar parcialmente o código do ambiente de verificação do DPCM. Para concluir o ambiente de verificação, o engenheiro vai precisar especificar alguns detalhes, por exemplo, a função utilizada para a geração dos estímulos aleatórios. No caso do componente DUV, a ferramenta vai gerar a sua "casca" a partir das informações sobre sinais e hierarquia extraídas dos códigos 1 e 2. A linguagem para a geração deste DUV parcial é SystemC.

Código 1 : Descrição do tipo de comunicação da TLN

```
01 // Specifications for the DPCM converter.
02         // dpcm sample
03         struct <DPCM_DATA> {
04
05           trans {
06             signed [4] dpcmData;
07           }
08
09           signals{
10             signed [4] dpmcSgnl;
11             //signals for the handshake between comunicating blocks
12             bool valid;
13             bool inv ready;
14           }
15         }
16
17         // internal dpcm sample
18         struct <DPCM_UNSAT> {
19           trans {
20             signed [5] dpcmData;
21           }
22
23           signals{
24             signed [5] dpmcUnsaturedSgnl;
25             //signals for the handshake between comunicating blocks
26             bool valid;
27             bool inv ready;
28           }
29         }
```

Código 2 : Descrição da hierarquia e de interfaces do DPCM.

```
30
31          //module to perform the difference between two dpcm samples
32          module DIF{
33            input dpcmSample difIn;
34            output dpcmUnsaturated difOut;
35          }
36
37          //module to perform the saturation
38          module SAT{
39            input dpcmUnsaturated satIn;
40            output dpcmSample satOut;
41          }
42
43          //the DPCM module
44          module DPCM{
45            // main module interfaces
46            input dpcmSample sampleIn;
47            output dpcmSample sampleOut;
48
49            // unsaturated fifo between modules
50            fifo dpcmUnsaturated difToSat;
51
52            // link between modules via difToSat fifo
53            DIF difI(.difIn(sampleIn), .difOut(difToSat));
54            SAT satI(.satIn(difToSat), .satOut(sampleOut));
55          }
```

2.6.3 O Componente *Source*

O componente Source é responsável por gerar os estímulos da verificação funcional. Estes estímulos são gerados no nível de transação. Quatro tipos de estímulos são recomendados por VeriSC: estímulos direcionados, estímulos de situações críticas, estímulos aleatórios e estímulos reais.

Os estímulos direcionados são escolhidos manualmente pelo engenheiro de verificação. Eles são interessantes em estágios iniciais da verificação funcional para se constatar a presença de erros crassos de projeto. Estímulos de situações críticas são aqueles responsáveis por estimular funcionalidades nas quais existe alguma suspeita de existência de erro. À medida que o número de funcionalidades do IP aumenta, torna-se impraticável para o engenheiro fazer a previsão dos estímulos que são capazes de exercitar as diversas partes do projeto do IP. Por esta razão, o Source deve ser capaz de gerar estímulos aleatórios que exercitem pontos do projeto que não foram antecipados pelo engenheiro de verificação. Por último, os estímulos reais são selecionados de uma situação real da aplicação do circuito digital em desenvolvimento.

A ferramenta eTBc gera a grande maioria do código do Source. Cabe ao engenheiro especificar a função para geração de valores aleatórios e os arquivos que contêm os estímulos direcionados, de situações críticas e reais.

2.6.4 O Modelo de Referência e o DUV

O Modelo de Referência é a implementação ideal do sistema. Assim, ao receber estímulos, ele deve produzir respostas corretas de acordo com a especificação do sistema. Naturalmente, a especificação deste componente vai mudar de acordo com o propósito do circuito digital a ser desenvolvido. Logo, não existe como prever um Modelo de Referência para cada sistema a ser desenvolvido. A ferramenta eTBc, no entanto, gera um molde que contém a "casca" do Modelo de Referência. Esta "casca" corresponde ao código que trata da comunicação com os componentes vizinhos, tais como entrada na FIFO dos dados que chegam do Source e saída dos dados na FIFO que são passados para o Checker. A partir deste molde, o engenheiro vai concluir o Modelo de Referência de acordo com a especificação do sistema.

VeriSC suporta Modelo de Referência em qualquer linguagem que seja capaz de se co-

municar com código SystemC no nível de transações, pois os demais elementos do ambiente de verificação também estão codificados nesta linguagem. Esta comunicação pode ocorrer através de FIFOs ou usando linguagem de programação que permita o acesso aos recursos de *pipe* do sistema operacional. Atualmente, o molde do Modelo de Referência é gerado também em SystemC.

O DUV não faz parte do ambiente de verificação. O DUV é o projeto sendo verificado. Como ele se comunica com o ambiente de verificação, é importante que sua implementação esteja de acordo com este ambiente desde o seu início. A implementação do DUV, que não considera o ambiente de verificação e vice-versa, pode levar a retrabalho para adequação de uma das partes. Da mesma maneira que o Modelo de Referência, não é possível prever o conteúdo do DUV. A aplicação alvo do sistema é quem vai orientar a composição desse módulo. Conseqüentemente, a automação da geração do DUV está restrita à geração de uma "casca" correspondente à parte de comunicação com o ambiente de verificação. Como este componente opera no nível de sinais e o Modelo de Referência opera no nível de transações, a parte de comunicação do DUV corresponde à geração de elementos de comunicação com os componentes TDriver e TMonitor. A linguagem do código gerado é SystemC.

2.6.5 Os Componentes TDriver e TMonitor

Um TDriver é responsável pela comunicação do Source com uma determinada interface de entrada do DUV. Uma interface de entrada é o canal pelo qual o DUV recebe dados de outros blocos. Assim, um ambiente de verificação pode possuir um ou mais componentes TDriver, dependendo da quantidade de interfaces de entrada do DUV. Cada interface de entrada possui seu protocolo de comunicação entre os blocos envolvidos. Este protocolo precisa ser parcialmente implementado pelo engenheiro de verificação.

O papel complementar ao do TDriver é executado pelo TMonitor. Assim, TMonitor tem o papel de converter estímulos na forma de sinais para transações. Igualmente, para cada interface de saída do DUV, vai existir um TMonitor. Em resumo, para o molde do TMonitor, o engenheiro de verificação precisa especificar as mesmas informações que são especificadas para o TDriver.

2.6.6 O Componente Checker

O Checker é responsável por comparar automaticamente os resultados advindos do Modelo de Referência e do DUV. Se valores diferentes são detectados na comparação, o engenheiro é informado desse erro mostrando-se para ele o valor esperado e o valor calculado pelo DUV. Esta informação é útil para auxiliar o engenheiro no processo de depuração. A geração automática deste bloco não necessita da intervenção do engenheiro, isto é, ela é realizada por completo pela ferramenta de geração de ambiente de verificação eTBc.

2.6.7 O Fluxo da Metodologia

O fluxo de atividades da metodologia VeriSC trata da implementação do ambiente de verificação antes da implementação do DUV. Esta abordagem possui várias vantagens em relação à abordagem que espera a conclusão da implementação do DUV para depois implementar o ambiente de verificação. Detalhes sobre estas vantagens podem ser encontrados nos trabalhos [32; 31]. A seguir, são citadas de maneira resumida algumas dessas vantagens:

1. O custo de integração do ambiente de verificação com o DUV é reduzido consideravelmente, podendo até ser nulo, pois o ambiente de verificação é gerado antes do DUV e a partir de uma especificação única para ambos. Esta especificação inicial única promove uma integração de interfaces entre as partes comunicantes do processo de verificação funcional.

2. A verificação funcional pode ser usada desde o início do projeto do DUV, economizando tempo de projeto. Sem o ambiente de verificação, o projetista do DUV vai ter que realizar testes durante o desenvolvimento do DUV ou então desenvolver todo o DUV para depois fazer a verificação funcional.

3. O ambiente de verificação pode passar por uma fase de depuração sistemática e bem elaborada de modo a diminuir significativamente a quantidade de erros nos elementos do ambiente de verificação.

4. Os elementos do ambiente de verificação podem ser reusados durante o seu desenvolvimento. Circuitos digitais são complexos e exigem mecanismos de decomposição de projeto. VeriSC define um fluxo que permite a decomposição dos projetos de modo

que os elementos desenvolvidos para a verificação funcional de cada parte de projeto sejam reutilizados na composição do ambiente de verificação funcional do projeto completo.

Primeiro passo: construção do ambiente de verificação completo A primeira atividade a ser realizada pelo engenheiro de verificação é a geração do ambiente de verificação para o DUV completo, isto é, sem decomposição. Esta atividade é constituída por outras três sub-atividades:

1. Teste de comunicação do Modelo de Referência com o ambiente de verificação. Para estimular o Modelo de Referência, um componente auxiliar, chamado de pré-source, é construído. Este componente auxiliar já considera a geração de estímulos direcionados e estímulos aleatórios. Para receber os resultados produzidos pelo Modelo de Referência, outro componente auxiliar, chamado de Sink, é construído.

2. Construção dos componentes Source e Checker. Neste passo, estes componentes também são validados. Para isso, o Modelo de Referência é replicado, sendo que a réplica faz o papel do DUV, que ainda não existe. Assim, o Source estimula os dois modelos de referência e o *Checker* verifica se os resultados produzidos são iguais. Adicionalmente, podem-se injetar erros na réplica do Modelo de Referência para analisar a habilidade do Checker de detectar tais erros.

3. Construção e teste dos componentes TDriver e TMonitor. Para realizar o teste sobre estes elementos sem a implementação do DUV, a organização da verificação funcional fica como ilustrado na Figura 2.4. A réplica do Modelo de Referência continua sendo usada, porém ela vai receber estímulos de uma réplica do componente TMonitor. Este TMonitor vai receber estímulos do TDriver original, convertendo-os da forma de sinais para a forma de transações. Na saída da réplica do Modelo de Referência, será colocada uma réplica do componente TDriver para viabilizar a comunicação com o componente TMonitor original.

Segundo passo: decomposição hierárquica do Modelo de Referência Neste passo, o engenheiro vai decompor o Modelo de Referência seguindo a mesma estrutura hierárquica

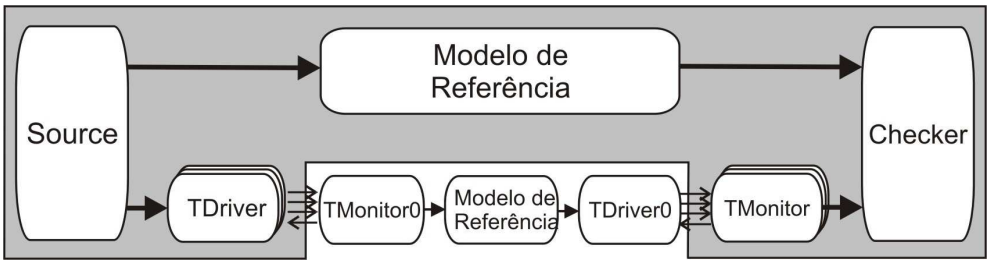

Figura 2.4: Etapa de construção do ambiente de verificação.

do DUV. Cada bloco resultante da decomposição hierárquica passa a ser tratado como um bloco independente, sendo que para cada bloco será constituído um ambiente de verificação. Porém, antes disso, a decomposição hierárquica é testada para se certificar de que nenhum erro foi inserido nessa fase. Este teste é feito fornecendo-se estímulos para os modelos de referência sem decomposição e com decomposição sendo a comparação dos resultados realizada pelo componente *Checker*.

Terceiro passo: ambiente de verificação para o DUV O terceiro passo trata de construir o ambiente de verificação para cada bloco da decomposição hierárquica do DUV. Para cada ambiente de verificação criado, devem ser seguidas as mesmas regras e os mesmos passos para construção do ambiente de verificação completo. Neste passo, os componentes TDriver e TMonitor do ambiente de verificação completo podem ser reusados, pois tais interfaces são mantidas na decomposição. As interfaces de comunicação entre blocos da decomposição hierárquica também podem ser reusadas para a construção do ambiente de verificação individual de cada bloco. Seguindo este princípio, a metodologia pode ser usada para qualquer número de blocos e qualquer topologia de decomposição hierárquica.

Quarto passo: substituição do DUV completo No quarto e último passo, todos os blocos que compõem o DUV são ligados em um único bloco. Em seguida, deve-se testar se erros não foram inseridos nesta ligação. O bloco completo resultante da ligação de cada sub-bloco do DUV é usado no ambiente de verificação criado no primeiro passo.

2.6.8 Análise de Cobertura em VeriSC

A metodologia VeriSC provê análise de cobertura através de uma biblioteca C++, chamada de *BVE-COVER (Brazil-IP Verification Extension*), que pode ser usada com as demais bibliotecas SystemC [78]. Como VeriSC é uma metodologia *black-box*, isto é, a verificação funcional não trata de detalhes internos do DUV, a análise de cobertura é realizada observando-se as interfaces disponíveis e também o Modelo de Referência.

O primeiro passo para se implementar a cobertura funcional é definir um modelo de cobertura. Este modelo é um documento que contém a especificação da implementação da cobertura funcional. Ao se construir um modelo de cobertura, o engenheiro deve estar ciente dos objetivos esperados durante a verificação funcional. Em geral, estes objetivos são relações de resultados que o dispositivo deve produzir, relação de resultados que o dispositivo não deve produzir e situações que devem ser ignoradas na análise de cobertura.

Um possível modelo de cobertura para o DPMC hierárquico está descrito na Tabela 2.2. Esta tabela registra os valores que devem ser alcançados durante a verificação funcional. Em particular, as duas variáveis principais do problema estão sendo observadas: a amostra de entrada e a amostra de saída. Limite Inferior é o valor mínimo que a diferença entre duas amostras consecutivas pode possuir. Limite Superior é o valor máximo que a diferença entre duas amostras consecutivas pode possuir. Embora não esteja registrado na Tabela 2.2, o engenheiro pode definir critérios de cobertura em relação aos valores que são passados entre os blocos comunicantes.

Tabela 2.2: Modelo de cobertura do DPCM.

Atributo	Amostra de Entrada	Amostra de Saída
Valores esperados	Limite Inferior - 1; Limite Inferior; Limite Inferior + 1; Limite Superior - 1; Limite Superior; Superior + 1	Limite Inferior; Limite Inferior + 1; Limite Superior; Limite Superior - 1

A biblioteca *BVE-COVER* é composta por quatro módulos diferentes: *BVE-COVER Bucket*, *BVE-COVER Illegal*, *BVE-COVER Ignore* e *BVE-COVER Cross-coverage*.

O módulo *BVE-COVER Bucket* é responsável por analisar as funcionalidades que devem ser cobertas na verificação funcional. As funcionalidades são especificadas como *baldes* que

devem ser preenchidos durante a simulação. Para evitar que a simulação fique eternamente como conseqüência de uma funcionalidade que de fato não existe no *design*, o tempo de simulação também pode ser usado como critério de parada para a simulação. As funcionalidades, assim como nos demais módulos de cobertura, são especificadas por instrumentação de código, tal como no exemplo do Código 3. Neste exemplo, está especificado que 10 amostras de saída do DPCM devem ser iguais ao limite superior de saturação e que outras 10 devem ser iguais ao limite inferior de saturação.

Código 3 : Instrumentação de código para análise de cobertura.

```
01          BVE_COVER_BUCKET Cv_bucket_Limit;
02
03          Cv_Bucket_Limit.begin();
04            BVE_COVER_BUCKET(Cv_bucket_Limit, (sampleOut == UPPER_LIMIT,10));
05            BVE_COVER_BUCKET(Cv_bucket_Limit, (sampleOut == LOWER_LIMIT,10))
06          Cv_Bucket_Limit.end();
```

O módulo *BVE-COVER Ignore* é responsável por analisar os *buracos* de cobertura. Um buraco de cobertura é um conjunto de funcionalidades que ficou descoberta durante a verificação funcional. Há dois tipos de buracos de cobertura: válido e inválido. Um buraco de cobertura válido é um requisito que faz parte do modelo de cobertura mas que não foi observado na verificação funcional. Um buraco de cobertura inválido é um requisito funcional que não deve fazer parte do modelo de cobertura. O módulo *BVE-COVER Ignore* é usado para evitar os buracos de cobertura inválidos, tornando mais precisa a análise de buracos de cobertura válidos.

O módulo *BVE-COVER Illegal* analisa a execução de funcionalidades que não estão de acordo com a especificação do sistema. Este módulo funciona como asserções que são inseridas no código e caso sejam executadas, elas são reportadas ao engenheiro de verificação.

A cobertura *Cross-coverage* é um modelo definido pela permutação de funcionalidades que devem ser exercitadas. Os baldes de *BVE-COVER Bucket* são usados para se construir matrizes multidimensionais. A cada novo evento, o simulador verifica se algumas dessas informações cruzadas ocorreram.

2.7 Trabalhos Relacionados

A fase em que diversos blocos mais simples são integrados para compor um sistema mais complexo é abordada de diversas maneiras nas metodologias de verificação funcional. De acordo com Keating [54], o plano de verificação de um IP complexo deve conter um conjunto de especificação de funcionalidades que devem ser verificadas em cada sub-bloco e outro conjunto de especificação de funcionalidades que devem ser verificadas para o bloco resultante da integração dos sub-blocos, de agora em diante, bloco macro. A partir destas especificações, o código RTL de cada sub-bloco é produzido e a verificação funcional realizada. Uma vez que todos os sub-blocos estão prontos, a integração de fato ocorre. A verificação funcional, no entanto, agora é realizada de acordo com as especificações do bloco macro. A qualidade de verificação funcional de cada sub-bloco depende do seu projeto. Para alguns sub-blocos uma verificação funcional de boa qualidade pode ser alcançada com ambientes de verificação simples. Para outros blocos, no entanto, uma boa verificação funcional é impossível sem se reproduzir todos os demais blocos do sistema que estão na sua vizinhança. Para esses casos mais complexos, o papel da verificação se restringe à verificação de interfaces e de funcionalidades básicas através de ambientes de verificação simples. A verificação mais detalhada do mesmo se torna uma conseqüência da verificação da integração. Keating não discute a relação entre as especificações de cada sub-bloco e a especificação do bloco macro.

Meyer [58] argumenta que os requisitos da verificação partem comumente da visão de sub-blocos para, em seguida, tratar da visão macro, que é resultante da integração dos mesmos. Uma vez que se tem uma compreensão de todos os requisitos, a ênfase passa a ser a identificação de elementos que são comuns à verificação de mais de um sub-bloco e ao bloco macro. Esta identificação dos elementos em comum é importante para evitar implementação redundante e assim economizar recursos do projeto a partir de reuso. O reuso discutido por Meyer se restringe aos componentes de verificação utilizados no projeto. Para este tipo de reuso, Meyer destaca a importância de modularidade, encapsulamento, interfaces bem definidas, abstrações e boas práticas de programação na definição destes componentes para que o reuso possa de fato economizar os recursos do projeto. Este reuso consiste em migrar os componentes utilizados na verificação de sub-blocos para a verificação do bloco macro. Meyer relata que nem todos os componentes vão migrar do sub-blocos para o bloco

macro. Por exemplo, os testes de sub-blocos que se concentram em seus cenários críticos permanecem no nível dos sub-blocos. Meyer comenta que o plano de verificação serve para identificar que componentes podem migrar dos sub-blocos para o bloco macro e não dos componentes que de fato devem migrar. Isto é, o foco do reuso é meramente estrutural e quantitativo dado que ele não prioriza a migração dos elementos críticos definidos em cada sub-bloco.

Bergeron [19] define que, no plano de verificação, o engenheiro deve identificar os requisitos da infra-estrutura necessários para se produzir um projeto com alto grau de probabilidade de ser livre de erros. Nesta fase, as partições de projeto que podem ser verificadas independentemente são identificadas. Este particionamento é realizado de modo a tornar a especificação de componentes individuais mais natural e mais fácil de integração. O engenheiro não deve decompor o problema para se balancear o custo da verificação funcional de cada componente. Bergeron destaca que níveis melhores de controlabilidade e observabilidade são alcançados em projeto menores, porém, partições menores significam um aumento no número de componentes de verificação e aumento de requisitos de verificação da integração dos mesmos. Para aumentar a produtividade, os componentes de verificação reusáveis devem ser identificados. Cada projeto verificado isoladamente apresenta um conjunto de interfaces que são pontos de comunicação com o ambiente de verificação. Um subconjunto destas interfaces estará presente na fase de verificação no nível do sistema. Assim, as interfaces que são compartilhadas por múltiplos projetos são oportunidades de reuso de componentes do ambiente de verificação. Assim como nas outras metodologias de verificação funcional apresentadas até então, esta metodologia de Bergeron não provê orientações para se arquitetar blocos e fazer integração dos mesmos de modo que o esforço gasto no desenvolvimento dos blocos não seja descartado na verificação da integração resultante. No entanto, em vez de serem diretamente controlados, os blocos na integração são reconfigurados para realizar operações relevantes ao sistema como um todo. Assim, os componentes de verificação devem ser reconfiguráveis para atender os propósitos da verificação no nível de blocos e no nível de sistema.

O que se percebe nas metodologias de verificação funcional na fase de integração de blocos de projetos é uma ênfase no reuso dos componentes do ambiente de verificação dos blocos envolvidos na integração. A estratégia de integração proposta neste documento visa,

além do reuso de componentes, melhorar a qualidade da verificação funcional, fazendo com que ela seja capaz de cobrir cenários complexos que não foram previstos pelo engenheiro de verificação. Sob esta perspectiva de melhoramento da verificação funcional, muitos outros trabalhos têm sido propostos.

De acordo com Moudanos *et al* [61], os erros difíceis de serem descobertos são aqueles que exigem uma seqüência complexa de interações entre as múltiplas sub-partes que compõem o projeto. Uma suíte de testes que cobre todas as possíveis transições do fluxo de controle pode minimizar o tempo de simulação e maximizar a probabilidade de encontrar erros difíceis. O grafo de fluxo de controle trata do seqüenciamento dos possíveis comandos do código RTL. Partindo-se desta premissa, o artigo apresenta um método automático para extrair o fluxo de controle de um circuito de modo que o espaço de estados resultante pode ser explorado para análise de cobertura e geração automática de teste. Assim, a idéia central do método é manter o foco no fluxo de controle do circuito.

O fluxo de controle é representado por uma máquina de estados finito. Esta máquina de estados é uma representação do comportamento do sistema, mas com número menor de estados em relação ao espaço de estados original do sistema. Esta redução é obtida basicamente desconsiderando-se as variáveis que não influenciam no fluxo de controle. Fica por conta do engenheiro a definição dos registradores que devem ser considerados como aqueles que influenciam no fluxo de controle do circuito. Adicionalmente, os registradores restantes são agrupados em classes de equivalência em relação ao efeito dos mesmos no fluxo de controle.

O espaço de estados resultante da abstração realizada é usado para análise de cobertura e geração de casos de testes. A suíte de testes original é usada para medir quanto do espaço de estados resultante foi coberto. A parte descoberta do espaço de estados é utilizada para a geração de novos testes. Isto é feito combinando-se ATPG (*Automatic Test Pattern Generation*) e técnicas de verificação formal [27], tais como diagramas de decisão binários dirigidos e interpretação abstrata, para gerar automaticamente as seqüências de estados que cobrem as partes que ficaram descobertas na simulação.

O trabalho de Moudanos *et al* é uma evolução do trabalho de Ho et *et al* [49], que apresenta uma técnica para avaliar a qualidade da cobertura funcional a partir de uma técnica de análise estática do código RTL. A análise estática elimina as variáveis que podem ser testadas de forma independente e mantém as interações que afetam somente o fluxo do caminho

de dados. O fluxo de caminho de dados é a representação da relação de dependência entre os dados do código RTL de acordo com o fluxo de controle do mesmo.

Uma vez eliminadas as variáveis que podem ser testadas de forma independente, o espaço de estados do código remanescente é construído e as métricas de cobertura são definidas sobre o mesmo. A estratégia reduz o número de transições que precisam ser cobertas mantendo a qualidade da métrica. O trabalho foi aplicado em parte de um projeto de um multiprocessador. O que se conclui dos resultados experimentais é que a importância do trabalho proposto não consiste em aumentar a cobertura, mas em fornecer ao engenheiro informações sobre quais testes importantes deixaram de ser realizados. De certa forma, este trabalho forma um dual com o artigo do trabalho de Moudanos *et al*. O trabalho Moudanos *et al* mantém o foco no fluxo de controle e vai mais além por tratar da síntese automática dos casos de testes para aumentar a cobertura.

As métricas de cobertura funcional tradicionais tratam quase que exclusivamente de processos individuais. No entanto, os projetos de hardware são compostos de vários processos que devem interagir de forma correta para implementar as especificações do sistema. Conseqüentemente, a ausência destas métricas faz com que a verificação funcional não trate de forma adequada erros que surgem quando componentes são integrados para compor o sistema. Diante deste cenário, os trabalhos de Harris [48] e Verma *et al* [86] apresentam uma métrica de cobertura que considera a interação entre processos. Eles modelam o comportamento de cada processo com um grafo de fluxo de controle e assume que executar todos os caminhos do fluxo de controle é suficiente para validar cada processo. Uma interação é modelada por um conjunto de caminhos em diferentes processos, executados em seqüência. No pior caso, o conjunto de potenciais interações é o produto cartesiano de conjuntos de caminhos dos processos individuais. O que pode ser um número demasiado grande para se computado. Este problema é resolvido identificando-se pares de caminhos que são inválidos porque a atribuição de valores no primeiro viola condições no fluxo de controle do segundo. Além disso, pares de caminhos são considerados como interações se o segundo é conseqüência direta do primeiro via sinais compartilhados. A idéia do trabalho, portanto, é construir um grafo com as possíveis interações entre dois processos e medir quanto deste grafo a verificação funcional está exercitando. O autor fez um estudo de caso em que falhas foram injetadas no RTL. Sobre este código RTL foi feita a verificação funcional usando-se métri-

cas de caminhos de cada processo individualmente e métricas de interação entre processos. Os resultados experimentais mostraram que, para se cobrir um percentual X de erros, é necessária uma cobertura menor de interação em relação à cobertura de caminhos. Em outras palavras, a cobertura de interações foi mais precisa para se estimar os erros injetados que as métricas de caminhos individuais.

Segundo Mishra *et al* [59], a verificação funcional é o principal gargalo no projeto de um microprocessador devido à falta de técnicas para estimar cobertura funcional. No referido artigo, os autores apresentam uma técnica de geração de testes para microprocessadores com *pipelines* baseada em cobertura funcional. O ponto principal do trabalho é o desenvolvimento de um modelo de grafos que registra estrutura e comportamento de uma grande variedade de processadores. A partir deste modelo, eles apresentam um modelo de faltas para ser usado na verificação funcional. Por último, eles apresentam procedimentos para geração de testes. A entrada destes procedimentos é o modelo de grafos e a saída são programas para testes que são capazes de cobrir todas as faltas do modelo de faltas. Os resultados experimentais mostraram que o número de testes gerados pela ferramenta é no mínimo a metade de testes gerados aleatoriamente para se alcançar a mesma cobertura. O diferencial do trabalho é o domínio da aplicação. A solução proposta se baseia em características especificas de processadores com *pipelines*.

Os trabalhos de Gupta *et al* [45], Banerjee *et al* [16] e IP [50] são exemplos de trabalhos que combinam técnicas de verificação formal com verificação funcional. A metodologia proposta por Gupta *et al* extrai um modelo abstrato do projeto para realização da verificação funcional. No entanto, o foco é a correção de propriedades em lógica temporal em vez de métricas de cobertura convencionais, tais como cobertura de código e funcional. Em seguida, as propriedades que são verificadas no modelo abstrato são usadas para guiar a simulação no ambiente de verificação funcional com o DUV de fato. Para lidar com o problema da explosão de espaço de estados [84], técnicas de interpretação abstrata e refinamento são empregadas [27]. Banerjee *et al* apresentam métodos para geração automática de testes a partir de especificações formais em lógica temporal. A geração automática funciona como um jogo entre o DUV e o ambiente de verificação. O objetivo DUV é satisfazer certa propriedade a partir dos resultados do seu processamento. O papel do ambiente de verificação é refutar tal propriedade a partir do controle das entradas que são produzidas para o DUV. O DUV tem

um erro se o ambiente de verificação tem uma estratégia que refute a propriedade. Ip [50] propõe uma técnica chamada TST (*Test Stimulus Transformation*), que incorpora idéias de verificação formal no ambiente de verificação. Na simulação tradicional, dados históricos sobre os cenários exercitados não são mantidos. Conseqüentemente, um simulador pode repetidamente exercitar os mesmos estados sem alcançar cenários críticos. TST faz uso de um espaço de estados da simulação para evitar trabalho redundante na simulação. Mesmo no caso da simulação, manter o espaço de estados em memória pode ser inviável. Assim, TST faz uso de técnicas de abstração e aproximação. Para melhorar a qualidade da simulação, TST faz modificações nos estímulos baseando-se nas informações registradas no espaço de estados.

2.8 Considerações Finais do Capítulo

Para melhorar a qualidade da verificação funcional, os trabalhos apresentados na seção anterior fazem uso de técnicas baseada em construção de espaço de estados ou outras técnicas formais. O objetivo do trabalho proposto também é melhorar a qualidade da verificação funcional, mas explorando uma fase específica de desenvolvimento, que é fase em que diversos blocos mais simples são integrados para compor um sistema mais complexo.

O que se percebe nas metodologias de verificação funcional na fase de integração de blocos de projetos é uma ênfase no reuso dos componentes do ambiente de verificação dos blocos envolvidos na integração. Esta ênfase pode ser constatada pela preocupação com modularidade, encapsulamento, interfaces bem definidas, abstrações e boas práticas de programação ao definir os componentes do ambiente de verificação. Esta preocupação, no entanto, não é suficiente para uma exploração efetiva dos cenários complexos que se configuram pela interação dos blocos. Desta forma, o objetivo do trabalho é sistematizar a verificação funcional na fase de integração de blocos de projeto de circuitos digitais para que novos cenários emergentes da interação entre blocos sejam explorados.

Capítulo 3

Verificação Funcional na Fase de Integração de Blocos de Projeto

3.1 Estratégia Básica de Integração

Em relação à abordagem de integração de blocos de projeto, algumas metodologias prescrevem a construção e verificação de todos os blocos para depois realizar a integração e verificação como um todo [54; 31]. Alguns autores definem esta estratégia como sendo do tipo *big bang* [53; 22]. Este tipo de integração apresenta problemas porque quando erros na integração ocorrem, o engenheiro tem poucos elementos para se orientar na localização e correção dos mesmos. Conseqüentemente, o tempo de integração pode aumentar, tornando a verificação mais cara. Para evitar problemas desta natureza, duas abordagens de integração são utilizadas: *top-down* e *bottom-up* [22; 53; 62]. Para explicar cada uma delas, será considerado um sistema hipotético definido segundo a hierarquia ilustrada na Figura 3.1.

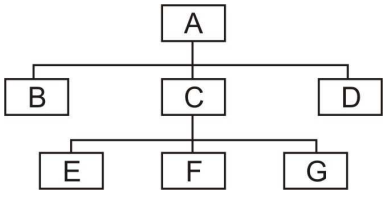

Figura 3.1: Hierarquia de módulos com três níveis e sete módulos.

Na abordagem *top-down*, o engenheiro, em geral, faz uso de *stubs* para proceder a veri-

ficação da integração. Um *stub* é um pedaço de software construído para simular partes do projeto que ainda não foram desenvolvidas, mas que são necessárias para verificar as demais partes projeto que dependem dele. A integração dos módulos A e B, ilustrada no quadro número 1 da Figura 3.2, requer o uso de *stubs* para os módulos C e D, representados por retângulos cinza. As interações entre A e B são concretas, conseqüentemente, o foco da verificação é o comportamento emergente da comunicação destes dois blocos. Em seguida, conforme ilustrado no quadro número 2, o *stub* D é substituído pela sua implementação de fato. Neste momento, a verificação deve focar em dois pontos: primeiro no teste de interface entre A e D, depois nos testes de regressão em busca de problemas na comunicação entre A e B na presença do módulo D. A substituição do *stub* C pela sua implementação de fato requer a inclusão dos *stubs* E, F, G, conforme ilustrado no quadro 3. Novamente, primeiro foca-se na verificação funcional entre C e os blocos que se comunicam com ele, para em seguida verificar o sistema como um todo na presença do novo bloco incluído na integração. E assim é conduzida a verificação na integração *top-down* até que se obtenha o sistema completo. O resto do processo da integração está ilustrado nos demais quadros da Figura 3.2.

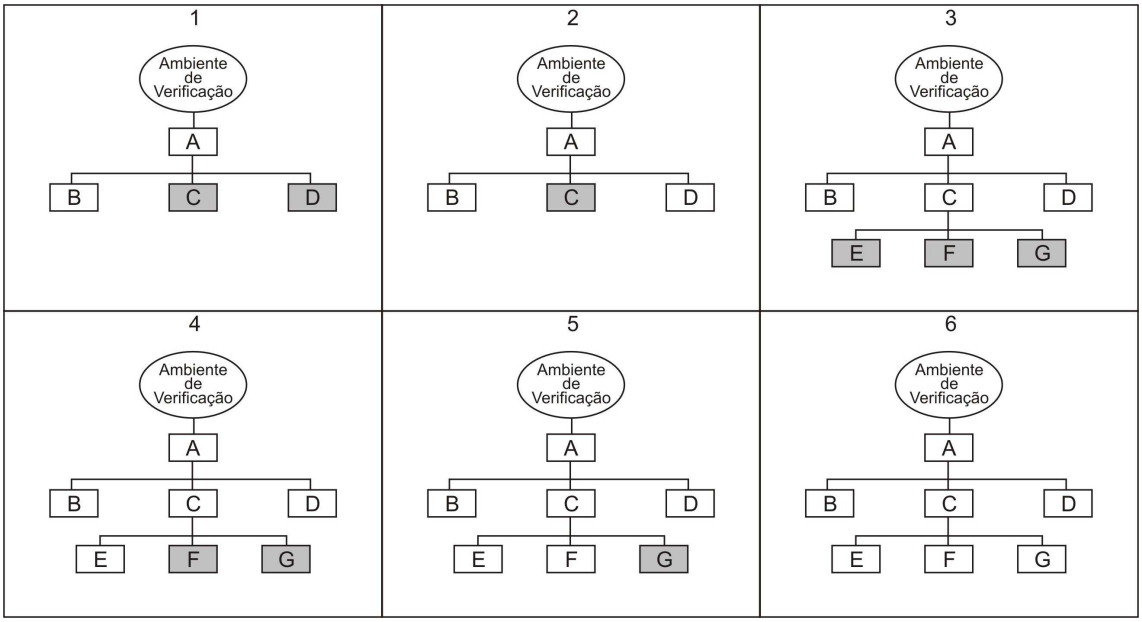

Figura 3.2: Verificação funcional com integração *top-down* de blocos de projeto.

Na abordagem *bottom-up*, a integração se inicia a partir dos blocos de mais baixo nível na hierarquia. Um bloco pode ser identificado como sendo de mais baixo nível se ele não depende de nenhum outro bloco abaixo dele para realizar sua computação. Diferentemente

da abordagem *top-down*, a abordagem *bottom-up* pode ser conduzida sem a utilização de *stubs*. No entanto, ao iniciar a integração é assumido que todos os blocos foram previamente validados. À medida que ocorre a integração, cabe ao ambiente de verificação fazer com que os blocos integrados sejam simulados como se eles estivessem operando no sistema completo. Uma vez que a verificação do sistema já integrado é considerada satisfatória pelo engenheiro, o ambiente de verificação corrente pode ser substituído pelo módulo logo acima na hierarquia. Este processo de substituição ocorre até que todos os blocos tenham sido integrados.

Um exemplo de integração *bottom-up* da hierarquia ilustrada na Figura 3.1 é apresentado na Figura 3.3. Os módulos de mais baixo nível são E, F e G. Conforme ilustrado no quadro 1 da Figura 3.3, o ambiente de verificação se encarrega de simular a integração destes três blocos. Idealmente, este ambiente de verificação deve invocar as funcionalidades dos blocos E, F e G da mesma forma como elas seriam invocadas pelo bloco C. Uma vez concluída a verificação desta etapa do processo, um novo bloco é integrado pela substituição do ambiente de verificação corrente pelo bloco C de fato e um novo ambiente de verificação é construído. Nesta fase, novos blocos de projeto no mesmo nível hierárquico de C podem ser integrados, tais como B e D. E o ambiente de verificação, neste caso, deve ser uma mímica do comportamento do bloco A (vide quadro 2 da Figura 3.1). Por último, no quadro 3, o ambiente de verificação é substituído pelo bloco A, concluindo assim a inclusão de todos os blocos do projeto. Agora a preocupação do engenheiro é fazer com que o ambiente de verificação simule os cenários nos quais o sistema será utilizado.

A abordagem *top-down* tem a vantagem de permitir que o engenheiro observe o comportamento do sistema como um todo desde o início da integração. Porém, a qualidade dos *stubs* pode determinar que certas funcionalidades do sistema sejam observadas somente com a implementação concreta do bloco de projeto. Na abordagem *bottom-up*, o engenheiro pode suprimir a construção de *stubs*. Porém, as funcionalidades do sistema só podem ser verificadas de fato após a inclusão de todos os blocos.

A proposta original de VeriSC [31] não define uma abordagem de verificação funcional na fase de integração dos blocos do projeto. No projeto do decodificador de vídeo MPEG 4 [69], por exemplo, foi utilizada uma abordagem *big-bang*. Conforme comentado anteriormente, este tipo de integração apresenta problemas porque quando erros na integração

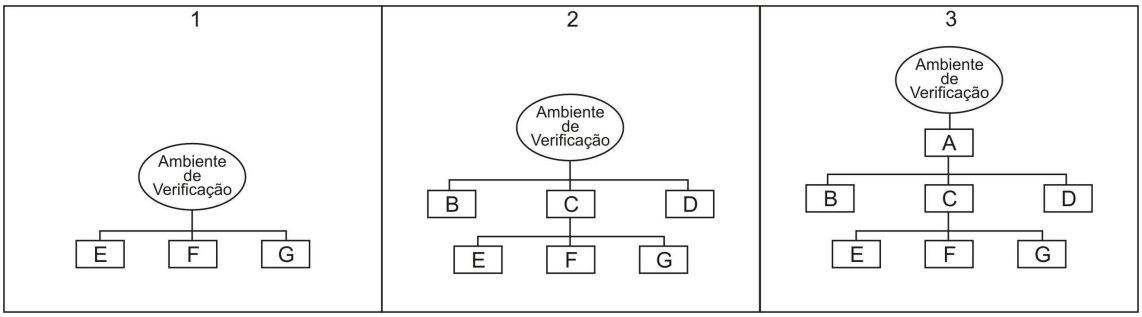

Figura 3.3: Verificação funcional com integração *bottom-up* de blocos de projeto.

ocorrem, o engenheiro tem poucos elementos para se orientar na localização e correção dos mesmos. Conseqüentemente, o tempo de integração pode aumentar, tornando a verificação mais cara. Já nas abordagens *bottom-up* e top-down, a identificação de erros se torna mais simples porque a inclusão de novos blocos de projeto se dá de forma incremental e sistemática.

A constituição dos ambientes de verificação em VeriSC, no entanto, se dá através de uma abordagem *top-down*. Conforme apresentado no Capítulo 2, a verificação funcional pressupõe a existência do modelo de referência do sistema devidamente hierarquizado de acordo com as especificidades do projeto. A partir deste modelo de referência, o ambiente de verificação do sistema como um todo é construído. Em seguida, ocorre uma decomposição deste ambiente de verificação para a construção do ambiente de verificação dos blocos de projeto que compõem o sistema. Na construção dos ambientes de verificação do sistema como um todo e dos blocos que o compõem, o modelo de referência serve como *stub* para a verificação de que os componentes que compõem o ambiente de verificação funcionam conforme esperado. O modelo de referência atua como DUV para verificar, entre outras coisas, se a conversão de estímulos na forma de transações para forma de sinais está correta. Depois que todos os ambientes de verificação são construídos e atestados de que operam conforme esperado, inicia-se a implementação ou reuso do DUV de fato.

A verificação da integração dos blocos de projeto em VeriSC pode ocorrer de forma *top-down* ou *bottom-up*. No caso de ser *top-down*, os modelos de referência dos blocos da hierarquia podem funcionar como *stubs* da mesma forma em que eles foram utilizados para a construção dos ambientes de verificação. No entanto, como a ênfase deste trabalho é obter cenários complexos que podem surgir da interação entre os blocos que compõem o

sistema, será assumida uma integração *bottom-up* em VeriSC. A idéia básica é melhorar a especificação do ambiente de verificação à medida que cada bloco de projeto é integrado. Além disso, será adotada uma estratégia de integração *bottom-up* que é realizada bloco a bloco, até que se obtenha todo o sistema. De fato, o engenheiro pode compor mais de dois blocos por vez, mas isto não é recomendado porque quando um erro é observado, o custo para depurá-lo pode ser maior. A integração bloco a bloco a ser utilizada neste trabalho está ilustrada na Figura 3.4. Existem dois blocos, X e Y, em que os resultados produzidos na saída de X são utilizados para alimentar o bloco Y. Nós estamos usamos o símbolo $\circ$ para representar este tipo de integração.

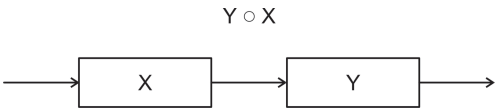

Figura 3.4: Dois blocos compondo um sistema hipotético.

De acordo com Ye [91], à medida que a complexidade de um projeto cresce, a complexidade da tarefa de verificação funcional deste projeto aumenta exponencialmente. Neste contexto, o reuso dos componentes do ambiente de verificação pode reduzir consideravelmente o esforço para se construir ambientes de verificação. Assim sendo, nossa proposta de integração de blocos de projeto também deve considerar o reuso dos componentes do ambiente de verificação. Na Figura 3.5 é apresentada a organização do ambiente de verificação na integração de dois blocos conforme ilustrado na Figura 3.4. Nas partes a) e b) desta figura, tem-se o ambiente de verificação para os blocos X e Y respectivamente. Na parte c), tem-se o ambiente de verificação para a integração Y$\circ$ X. O ambiente de verificação da integração é constituído em sua totalidade por componentes que foram previamente utilizados na verificação funcional do bloco X ou do Y. Da verificação funcional do bloco X, são reusadas as partes de geração de estímulos, *TDriver* e modelo de referência. Da verificação do bloco Y, são reusados os componentes *TMonitor*, *Checker* e modelo de referência. Por ser baseada em reuso, esta organização proposta promove uma redução significativa do custo de se construir um ambiente de verificação. A cada bloco de projeto integrado, o engenheiro não precisa construir um ambiente de verificação a partir do zero. O que ele precisa fazer é configurar um novo ambiente de verificação para que os componentes previamente desenvolvidos sejam organizados conforme ilustrado na Figura 3.5.

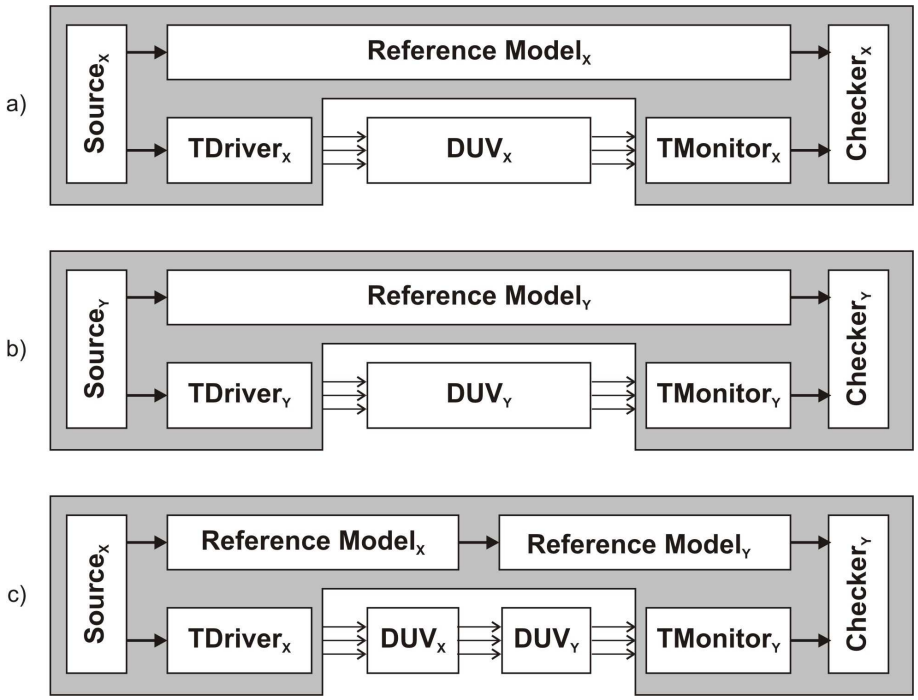

Figura 3.5: Exemplos de ambientes de verificação

3.2 Detectando Novos Cenários de Simulação

A configuração de geradores de estímulos e a especificação de critérios de cobertura são atividades que dependem fortemente da intervenção direta do engenheiro. Estas atividades são influenciadas, entre outras coisas, pela complexidade do sistema em desenvolvimento e por pressões de mercado para que o projeto seja concluído em um prazo que não comprometa a comercialização do produto final. Esta componente humana faz com que a efetividade da verificação funcional sofra variações de projeto para projeto [26; 88]. Conseqüentemente, no momento de se realizar a integração de um bloco recém-criado ou reusado de outro sistema, vários pontos, tais como comportamento, interface e protocolo de comunicação, precisam ser devidamente tratados para que o sistema final se comporte conforme determina sua especificação [60].

A organização do ambiente de verificação conforme discutido na seção anterior é fortemente baseado no reuso dos componentes do ambiente de verificação. No entanto, este reuso não é suficiente para garantir que cenários que foram explorados nos blocos individualmente serão explorados na integração. Este fato pode limitar a detecção de erros que surgem em função da interação entre os blocos integrados. Por exemplo, no item c) da Figura 3.5, os

critérios de cobertura que existiam no bloco $TMonitor_X$ do item a) deixam de existir. Assim, a verificação funcional da integração pode ser concluída sem que as funcionalidades especificadas e verificadas no bloco X em separado sejam propagadas para o bloco Y durante a verificação da integração.

Outro problema que pode acontecer em decorrência do reuso estrutural dos componentes de verificação é a caracterização de critérios de cobertura que não são alcançáveis em um tempo aceitável. Considerando-se o cenário de integração ilustrado na Figura 3.5, a verificação funcional pode resultar em um cenário conforme ilustrado no gráfico da Figura 3.6. Esta integração foi extraída do projeto do decodificador de vídeo MPEG 4 [69]. Os blocos DCDCT e IS quando verificados individualmente alcançam a cobertura esperada, mas a verificação funcional da integração não atinge 100 % em um tempo aceitável. Como a observabilidade e controlabilidade tendem a diminuir à medida que os blocos de projeto são integrados para formar blocos mais complexos, cenários como esses podem se tornar um impasse para o engenheiro. Isto porque o engenheiro não dispõe de mecanismos para concluir se os critérios de coberturas são de fato inalcançáveis porque o bloco DCDCT restringe o conjunto de estímulos fornecidos ao bloco IS, ou se o componente gerador de estímulos do bloco DCDCT não foi configurado de modo a gerar a entrada necessária para satisfazer critérios do bloco IS. Este impasse é bastante caro para o projeto. Ele pode provocar a aprovação de um projeto que não foi devidamente verificado, pois certos critérios de cobertura não foram atingidos, ou então há um desperdício de recursos na tentativa de se atingir critérios de cobertura que não são alcançáveis.

Para caracterizar as situações em que os problemas discutidos até então nesta seção podem ocorrer, vamos assumir que o ambiente de verificação na metodologia VeriSC é definido da seguinte maneira:

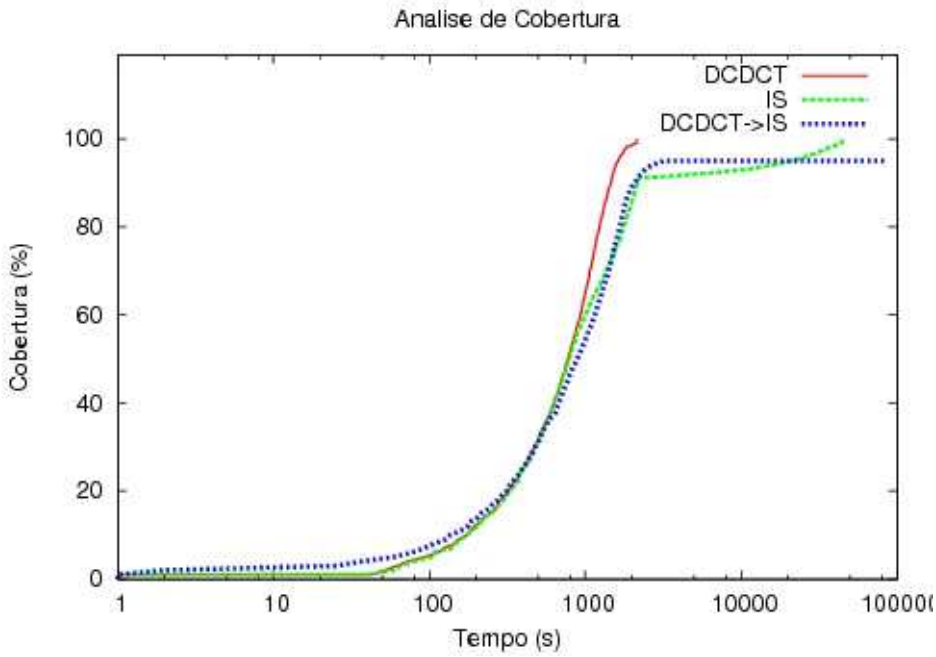

Figura 3.6: Progressão da análise de cobertura em função do tempo.

Definição 3.1 (Ambiente de Verificação) *Sejam* $\mathbb{I}$ *e* $\mathbb{O}$ *os conjuntos de todas as possíveis entradas e saídas respectivamente do circuito digital no nível de transações. Sejam também* $\mathbb{I}'$ *e* $\mathbb{O}'$ *os conjuntos de todas as possíveis entradas e saídas respectivamente do circuito digital no nível de sinais. O ambiente de verificação* T *é uma tupla* $T = \{Source, TDriver, RM, TMonitor, Checker\}$ *onde* $Source$ *é conjunto de estímulos,* $Source \subseteq \mathbb{I}$*;* $TDriver$ *é uma função* $TDriver : \mathbb{I} \longrightarrow \mathbb{I}'$*;* RM *é uma função* $RM : \mathbb{I} \longrightarrow \mathbb{O}$*;* $TMonitor$ *é uma função* $TMonitor : \mathbb{O}' \longrightarrow \mathbb{O}$ *e* $Checker$ *uma função* $Checker : \mathbb{O} \times \mathbb{O} \longrightarrow \{true, false\}$.

Sejam X e Y dois blocos de projeto que podem ser integrados conforme ilustrado na Figura 3.4, o ambiente de verificação da integração Y ∘ X pode ser definido conforme ilustrado na Figura 3.5. Neste caso, os componentes $Source_X$ e $TDriver_X$, pertencentes ao bloco X, são reusados. Seguindo o mesmo princípio, os componentes $TMonitor_Y$ e $Checker_Y$ são reusados do ambiente de verificação do bloco Y. Como resultado, a qualidade da verificação funcional é determinada por estes componentes que foram reusados. Estes componentes, no entanto, foram projetados para a verificação de cada bloco em separado. Além disso, o componente $TMonitor_X$ do ambiente de verificação do bloco X e os componentes $Source_Y$ e

$TDriver_Y$ do bloco Y não são usados na verificação da integração.

Antes de descartar os componentes que não foram reusados na integração, podemos utilizá-los para avaliar se cada bloco foi satisfatoriamente verificado separadamente. O componente $TMonitor_X$, usado para verificar o bloco X, pode conter especificações de cobertura definidas pelas declarações de BVE-COVER Bucket, BVE-COVER Illegal e BVE-COVER Ignore (vide Capítulo 2 para mais informações sobre especificação de cobertura em VeriSC). Vamos considerar que estas especificações constituem os conjuntos de valores A, definido por $A = \{$ BVE-COVER Illegal$\} \cup \{$BVE-COVER Ignore$\}$, e B, definido por $B = \{$BVE-COVER Bucket$\}$. O componente $Source_Y$, usado para estimular o bloco Y, contém especificações para a geração de estímulos que são usados para simular o bloco. A geração de estímulos aleatórios é baseada nas funcionalidades providas pelas classes `scv_bag` e `SCV_CONSTRAINT` de SystemC. Assim, podemos considerar que o componente $Source_Y$ também especifica um conjunto de valores C tal que $C = \{$`scv_bag`$\} \cup \{$`SCV_CONSTRAINT`$\}$.

Fazendo a interseção dos conjuntos A, B e C, nós podemos obter informações que podem servir para melhorar a especificação da verificação funcional de cada um dos blocos da integração.

Lema 1 *Sejam X um bloco de projeto e T_x o seu respectivo ambiente de verificação de acordo com a Definição 3.1. Seja A o conjunto de valores definido pelas especificações de BVE-Cover Bucket do componente $TMonitor_x$. Seja B o conjunto de valores definido por especificações do tipo BVE-Cover Illegal ou BVE-Cover Ignore do $TMonitor_x$. A expressão $A \cap B = \emptyset$ é válida.*

Prova: assumindo que existe pelo menos um elemento x que pertence ao conjunto A e ao conjunto B, isto é, $x \in A \cup B$ este elemento é simultaneamente um valor esperado na saída de X e um valor que não deve ser considerado ou ilegal, constituindo assim uma contradição. Portanto, os conjuntos A e B são conjuntos disjuntos, $A \cap B = \emptyset$. $\square$

De acordo com o Lema 1, podemos propor um diagrama de Venn conforme ilustrado na Figura 3.7. Analisando caso por caso, o conjunto D, $D = A - C$, é o conjunto de valores que são ilegais ou que não precisam ser considerados na saída do bloco X, e que também não aparecem na especificação dos estímulos de entrada do bloco Y. De fato, não é

Figura 3.7: Diagrama de Venn para os conjuntos de valores definidos pelos componentes de verificação.

um problema que entrada de Y não considere de valores que são ilegais ou que não precisam ser considerados na saída do bloco X. Se eles são indesejáveis ou irrelevantes na saída de X, podemos assumir que, para esta integração, a simulação de Y não depende de tais valores. No entanto, os problemas começam se o conjunto E, definido por $E = A \cap C$, não for vazio. Neste caso, a análise de cobertura da integração pode alcançar menos que 100% porque o conjunto E contém valores que são, ao mesmo tempo, valores esperados para simular o bloco Y e valores considerados ilegais ou irrelevantes para a saída de X. Assim, se pelo menos uma especificação de cobertura de Y depende de estímulos que pertencem ao conjunto E, esta cobertura poderá não ser satisfeita para a composição.

O conjunto F, $F = (C - A) - B$, contém valores que não são ilegais ou irrelevantes para o bloco X, e que são estímulos para o bloco Y. Além disso, estes valores não foram especificados como valores importantes que devem ser providos pelo bloco X porque eles não pertencem ao conjunto de valores definido por B. Assim, nós temos um buraco na especificação na saída do bloco X porque F contém valores especificados para estimular o bloco Y, mas que não se sabe se foram cobertos na simulação do bloco X separadamente.

Teorema 3.1 *Sejam X e Y dois blocos de projetos, sejam T_x e T_y seus respectivos ambientes de verificação de acordo com a Definição 3.1. Seja A o conjunto de valores definido por especificações BVE-Cover Illegal ou BVE-COVER Ignore contidas em $TMonitor_x$. Seja B o conjunto de valores definido por especificações BVE-Cover Bucket também contidas em $TMonitor_x$. Seja C o conjunto de valores definido por especificações de $Source_y$. Na composição $Y \circ X$, se o conjunto $C - (A \cup B) \neq \emptyset$, então as especificações de BVE-Cover Bucket de $TMonitor_x$ possuem um buraco.*

Prova: vamos assumir que BVE-Cover Bucket é completa em relação à composição, então temos que todos os estímulos necessários para alcançar os critérios de cobertura contidos

CONJUNTO	DESCRIÇÃO	ANÁLISE
A	Conjunto de valores que são ilegais ou que devem ser ignorados na saída do bloco X.	São valores especificados pelo engenheiro e que servem de base para obtenção de novas especificações.
B	Conjunto de valores que são especificados como critério de cobertura na saída do bloco X.	
C	Conjunto de valores que são utilizados na simulação do bloco Y.	
D	Resultados ilegais ou inesperados na saída X e que não aparecem na entrada de Y.	Representa uma conformidade entre a verificação funcional de cada bloco. Não representa problemas.
E	Valores esperados na entrada de Y e que são ilegais ou irrelevantes para a saída de X.	Representa uma desconformidade entre a verificação de cada bloco. Estes valores podem levar a cobertura inalcançável na integração.
F	Valores esperados na entrada de Y e que não foram considerados na saída de X.	Representa uma desconformidade entre a verificação de cada bloco. Estes valores representam cenários importantes que deveriam ser considerados na saída de X.
G	Valores esperados na entrada de Y e que foram considerados na saída de X.	Representa uma conformidade entre a verificação funcional de cada bloco. Não representa problemas.
H	Valores esperados na saída de X e que não foram considerados na entrada de Y.	Representa uma desconformidade entre a verificação de cada bloco. Estes valores representam cenários importantes que deveriam ser considerados na entrada de Y.

Tabela 3.1: Análise da informação descartada na integração.

em T_y foram considerados por T_x, isto é, $C \subseteq (A \cup B)$. De acordo com a teoria dos conjuntos, temos que $C - (A \cup B) = \emptyset$, que é uma contradição. □

Para preencher o buraco mencionando no Teorema 3.1, precisamos melhorar a especificação de cobertura na saída do bloco X. Este melhoramento pode ser realizado em A ou B. No entanto, como a verificação funcional tem como objetivo a detecção de erros, vamos assumir que as especificações de cobertura que estão faltando são alcançáveis e que devem ser consideradas durante a verificação funcional do bloco X. Assim sendo, precisamos melhorar a especificação do conjunto B.

O conjunto G, $G = C \cap B$, representa os valores que estão sendo considerados durante a verificação funcional do bloco X e do bloco Y. De fato, G é o conjunto de estímulos que são essenciais para X e Y. Portanto, o objetivo da integração deve ser a maximização do conjunto G porque ele representa a conformidade entre a verificação funcional dos blocos individualmente. Por último, o conjunto H, $H = B - C$, contém valores importantes que foram produzidos por X, mas que não foram utilizados na simulação do bloco Y porque eles não aparecem em C. Neste caso, temos que a verificação funcional do bloco Y tem um buraco em relação à integração sendo realizada. Para preencher este buraco, o componente $Source_Y$ deve ser especificado para considerar estes novos valores.

Teorema 3.2 *Sejam X e Y dois blocos de projeto, sejam T_x e T_y seus respectivos ambientes de verificação de acordo com a Definição 3.1. Seja A o conjunto de valores definido por especificações BVE-Cover Illegal ou BVE-COVER Ignore contidas em $TMonitor_x$. Seja B o conjunto de valores definido por especificações BVE-COVER Bucket também contidas em $TMonitor_x$. Seja C o conjunto de valores definido por especificações de $Source_y$. Na composição $Y \circ X$, se $B - C \neq \emptyset$, então a especificação de $Source_y$ tem um buraco.*
Prova: vamos assumir que a especificação de $Source_y$ está completa em relação à composição sendo realizada. Isto significa que todos os resultados especificados por T_x como sendo relevantes na saída de X são utilizados como estímulos por T_y, isto é, a expressão $B \subseteq C$ é válida. De acordo com a teoria dos conjuntos, temos que $B - C = \emptyset$, configurando assim uma contradição. □

3.2.1 Outras Topologias de Integração

Conforme mencionado na Seção 3.1, nós estamos considerando uma abordagem de composição em que um par de blocos de projetos é integrado por vez. A nossa discussão sobre uma integração em que dois blocos são combinados por vez se baseou no cenário ilustrado na Figura 3.4, no qual existe somente uma interface de comunicação entre os blocos X e Y. Para os sistemas em que há duas ou mais interfaces ligando os blocos X e Y, o engenheiro deve lidar com uma interface por vez, em processos separados, para a detecção de novas especificações. Isto é possível em VeriSC porque para todo bloco de projeto sob verificação D, para cada interface de entrada (saída), existe um componente TDriver (TMonitor). Assim sendo, os teoremas apresentados anteriormente são válidos para a composição em que há duas ou mais interfaces ligando o bloco X ao bloco Y. Para tanto, basta que o engenheiro trate cada interface de comunicação em separado.

Se considerarmos o diagrama de blocos do sistema como um grafo dirigido em que blocos de projetos são os vértices do grafo e as interfaces são setas indo de um vértice a outro, com dois vértices é possível ter apenas três grafos dirigidos diferentes [28], ilustrados na Figura 3.8. O caso a) da Figura 3.8 não é relevante para o nosso estudo porque nele não existe ligação entre os vértices. Assim, ele representa um projeto composto por dois blocos que não se comunicam entre eles. O caso b) é o diagrama ilustrado na Figura 3.4 e que foi usado como base para a discussão dos problemas que podem surgir na etapa de integração. O caso c) representa um projeto em que há uma interface do bloco X para o bloco Y e uma interface de comunicação entre o bloco Y e o bloco X. Para lidar com este cenário, em um momento o engenheiro deve considerar uma confrontação entres os componentes $TMonitor_X$ e $Source_Y$. Em outro, ele deve considerar uma confrontação entre $TMonitor_Y$ e $Source_X$.

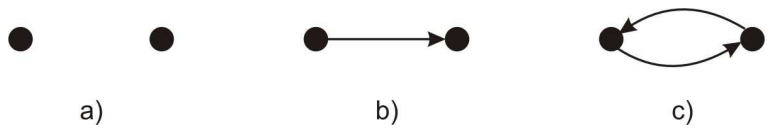

Figura 3.8: Possíveis grafos dirigidos com dois vértices.

3.3 Cálculo de Novos Cenários de Simulação

De acordo com os teoremas 3.1 e 3.2, os componentes que não são reusados durante a composição podem revelar cenários de simulação que não foram explorados durante a verificação funcional de cada bloco em separado. Para que estes cenários sejam considerados de fato, primeiro precisamos obter a especificações que faltam para cada bloco para, em seguida, configurar o ambiente de verificação de cada um dos blocos de projeto com estas novas configurações.

O conjunto G do diagrama de Venn da Figura 3.7 representa a conformidade entre as especificações dos ambientes de verificação dos blocos X e Y. Portanto, o que pretendemos é a maximização deste conjunto. A avaliação positiva da expressão $F = (C - A) - B \neq \emptyset$ ou $H = B - C \neq \emptyset$ determina que novos valores podem ser adicionados ao conjunto G.

A expressão $H = B - C \neq \emptyset$ significa que o resultado desta diferença deve se tornar especificação de geração de estímulos para a simulação do bloco Y. O resultado desta diferença representa os cenários de simulação que foram considerados na verificação funcional do bloco X e que devem ser propagados para a simulação do bloco Y. Isto significa que o componente $Source_Y$ deve ser reconfigurado para que estes novos valores sejam gerados. Neste momento, é importante observar que o reuso dos componentes do ambiente de verificação tal como definido na Seção 3.1 não é suficiente para garantir que, no momento da integração, estes valores serão propagados de X para Y. Pode ser o caso em que os critérios de cobertura que restaram no ambiente de verificação sejam satisfeitos sem que tais cenários sejam considerados em Y. No Capítulo 4, nós mostramos exemplos de integração no projeto do decodificador de vídeo MPEG 4 em que critérios de cobertura são satisfeitos na verificação funcional dos blocos individualmente, mas deixam de ser satisfeitos quando ocorre a integração.

A simples reconfiguração do $Source_Y$ para que os novos estímulos sejam gerados não é suficiente para garantir que tais estímulos serão de fato usados na verificação funcional. Pode ser o caso em que a cobertura funcional original de Y seja satisfeita sem que os novos estímulos tenham sido usados para simular o bloco de projeto. Para evitar este problema, os novos cenários obtidos precisam ser especificados como estímulos para o $Source_Y$ e critérios de cobertura para o $TDriver_Y$. Com estes novos componentes, o engenheiro pode realizar

a verificação funcional e a satisfação de 100% dos critérios de cobertura significa que os novos estímulos foram utilizados. Se a verificação funcional com estes novos componentes revelar algum erro, o engenheiro deve corrigi-lo para depois prosseguir com a composição hierárquica.

Os cenários de simulação que devem ser propagados do bloco Y para o bloco X são definidos pela expressão $F = (C - A) - B \neq \emptyset$. O resultado da diferença deve ser transformado em especificações de cobertura funcional que irão compor o componente $TMonitor_X$, juntamente com as especificações originais que já existiam para este bloco. Estas especificações são definidas como declarações do tipo BVE-Cover Bucket. Este novo $TMonitor_X$ exige que os novos cenários sejam exercitados na verificação funcional. Caso eles não sejam satisfeitos, a verificação funcional tem que ser encerrada sem que os critérios de cobertura tenham sido 100 % satisfeitos. É importante observar que reconfigurar o $TMonitor_X$ não significa que os novos cenários serão automaticamente alcançados. Pode ser o caso em que a especificação de geração estímulos do componente $Source_X$ seja inadequada para que os novos critérios de cobertura sejam alcançados. O trabalho proposto não trata da configuração automática de geradores de estímulos. Este problema está fora do escopo do nosso trabalho. Existe uma área específica que trata deste problema, chamada de Coverage Directed Test Generation (CDG) [37; 76]. No nosso trabalho, o engenheiro deve configurar manualmente os geradores de estímulos para que os novos critérios de cobertura introduzidos sejam satisfeitos.

Por último, se o conjunto E, definido por $E = A \cap C$, não for vazio, o engenheiro deve ser avisado de que a análise de cobertura da integração pode alcançar menos que 100 % porque existem valores que são, ao mesmo tempo, valores esperados para simular o bloco Y e valores ilegais ou irrelevantes na saída do bloco X.

3.4 Sistematização da Verificação Funcional na Fase de Integração

Esta seção é dedicada à apresentação do fluxo de trabalho que o engenheiro deve seguir para aplicar os resultados apresentados nas seções anteriores. Além disso, o suporte ferramental necessário para o seguimento de tal fluxo é discutido.

3.4.1 O Fluxo de Trabalho na Integração

Conforme apresentado no Capítulo 2, o fluxo de trabalho da metodologia VeriSC é definido a partir de quatro passos que devem ser seguidos pelo engenheiro de verificação. De fato, cada passo é constituído por um conjunto de atividades específicas. O primeiro passo é dedicado à construção do ambiente de verificação para o sistema completo, sem considerar a decomposição. No segundo passo, o engenheiro realiza a decomposição do sistema completo em blocos menores e constrói o ambiente de verificação para cada um deles. Nos dois primeiros passos, o ambiente de verificação é construído considerando-se somente o modelo de referência, sem precisar do DUV. Assim, a ênfase é a construir todos os ambientes de verificação e atestar que cada um deles opera conforme esperado. No terceiro passo, o engenheiro implementa o DUV para cada bloco e realiza a verificação funcional. No quarto e último passo, todos os blocos que compõem o DUV são ligados em um único bloco. O DUV resultante da ligação é verificado usando-se o ambiente de verificação desenvolvido no primeiro passo da metodologia.

O quarto passo da metodologia VeriSC é dedicado a fazer a integração dos blocos para compor o sistema completo. Portanto, a abordagem de integração proposta neste trabalho deve ser aplicada neste último passo. Conforme discutido na Seção 3.1, o engenheiro deve começar a integração por blocos de mais baixo nível na hierarquia, adicionando um bloco por vez na integração. Para cada bloco integrado ao sistema, o engenheiro analisa os componentes do ambiente de verificação que não serão reusados na verificação da integração. O resultado desta análise é um relatório reportando que as especificações dos dois blocos estão em conformidade ou então conjunto de especificações que devem ser incorporadas aos blocos da integração. Caso as especificações dos dois blocos estejam em conformidade umas com as outras, os dois blocos podem ser integrados e verificados. Caso novas especificações tenham sido reportadas, tais especificações são incorporadas ao ambiente de verificação de cada bloco sendo integrado. Em seguida, o engenheiro deve realizar a verificação funcional de cada bloco isoladamente para as novas especificações sejam consideradas. Uma vez que cada bloco foi verificado isoladamente com as novas especificações obtidas a partir do seu bloco vizinho, o engenheiro pode realizar a integração de fato e um novo ambiente de verificação é obtido a partir do reuso dos componentes de verificação desenvolvidos para cada bloco.

Alternativamente, o engenheiro pode optar por outro fluxo possível para a aplicação da abordagem proposta neste trabalho. Neste fluxo alternativo, o engenheiro pode continuar a realizar a integração *big bang*, tal como na proposta original de VeriSC. A diferença é que, antes de integrar todos os blocos, o engenheiro deve executar o algoritmo descrito no Algoritmo 1. A idéia do referido algoritmo é analisar cada interface de comunicação entre os blocos do projeto em busca de novas especificações de cobertura funcional. Porém, esta análise é realizada sem efetuar a integração de fato. A integração ocorre depois que a especificação de cada bloco de projeto foi analisada em relação à especificação de cada um dos blocos vizinhos. Em relação ao fluxo descrito anteriormente, este fluxo é recomendado para os casos em que o engenheiro não dispõe de suporte ferramental para a geração de ambientes de verificação funcional a cada momento em que um bloco é integrado ao sistema.

Algoritmo 1 : Atividades de verificação na fase de integração de blocos.

```
1: Para cada bloco de projeto X faça
2:     Para cada bloco de projeto Y na saída de X faça
3:         obtenha os componentes que são descartados na integração Y ∘ X
4:         analise os componentes que são descartados na integração Y ∘ X
5:         Se existem novas especificações para X então
6:             obtenha as novas especificações para X
7:             realize a verificação funcional de X com as novas especificações
8:         Fim
9:         Se existem novas especificações para Y então
10:            obtenha as novas especificações para Y
11:            realize a verificação funcional de Y com as novas especificações
12:        Fim
13:    Fim
14: Fim
15: Integre todos os blocos de projeto
```

3.4.2 Suporte Ferramental

Do ponto de vista de automação, o fluxo de atividades descrito anteriormente requer ferramentas para dois problemas distintos. O primeiro problema diz respeito à construção do

ambiente de verificação para os blocos de projeto que estão sendo integrados. Seguindo a mesma idéia da ferramenta eTBc [64], o engenheiro precisa de suporte para gerar de forma automática os componentes do novo ambiente de verificação. O outro problema diz respeito ao melhoramento das especificações propriamente.

Protótipo para Obtenção de Novos Critérios de Cobertura Funcional

A Figura 3.9 é uma ilustração de como funciona o protótipo para obtenção de novos cenários de simulação. Para a integração $Y \circ X$, as entradas do processo são os componentes TMonitor_X, Source_Y e TDriver_Y. Se for o caso de algum melhoramento em alguns destes componentes, eles são gerados como saída do processo. O protótipo que se encarrega de tal melhoramento tem dois módulos básicos: um módulo que funciona como *parser* e gerador de componentes de verificação e outro módulo que resolve expressões em teoria dos conjuntos conforme discutido na Seção 3.3.

Embora os conjuntos A, B e C sejam definidos originalmente como comandos em SystemC, cada um deles tem um propósito específico. Os elementos do conjunto B, por exemplo, são usados como critérios de cobertura, enquanto os elementos do conjunto C são usados para a geração de estímulos. Conseqüentemente, a sintaxe SystemC de cada um deles se difere. Esta diferenciação requer que as expressões que os definem sejam traduzidas para especificações abstratas que podem ser manipuladas por um módulo de teoria de conjuntos. Esta tradução é realizada pelo módulo *parser* e gerador de componentes, que lê os arquivos de entrada para construir as estruturas de dados necessárias para o cálculo de novos cenários de simulação. O cálculo de novos cenários é realizado pelo módulo solucionador de teoria de conjuntos. Caso existam novos cenários, os novos componentes são gerados de acordo pelo módulo *parser* e gerador de componentes, de acordo com a sintaxe SystemC de cada um dos conjuntos.

Nas fases de tradução de comandos SystemC, para obtenção dos conjuntos e tradução de conjuntos para comandos SystemC, o grande desafio foi lidar com o conjunto C, que diz respeito à geração de estímulos a partir da biblioteca SCV (*SystemC Verification Extensions*). A dificuldade em lidar com este conjunto é decorrente da diversidade de opções disponíveis para a geração de estímulos. A biblioteca SCV dá suporte a três modalidades de geração aleatória de valores para a verificação funcional: geração aleatória sem restrição, geração

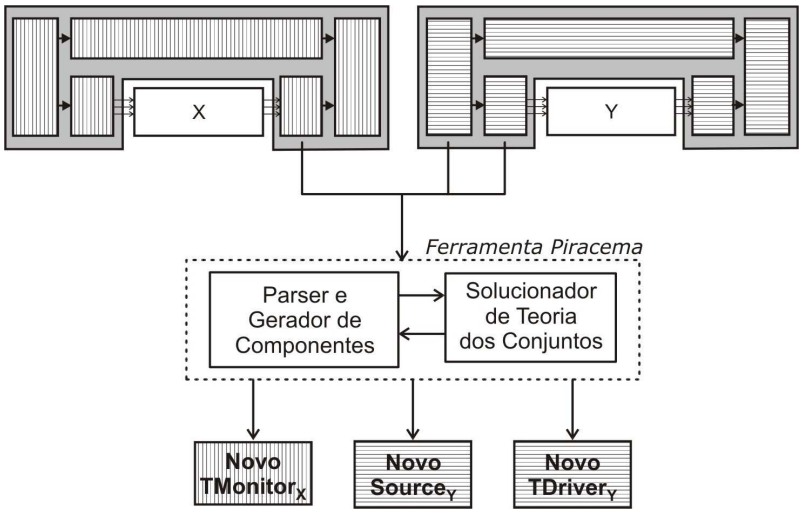

Figura 3.9: Visão geral da ferramenta para obtenção de novos cenários de simulação.

aleatória com restrição e geração aleatória com peso [79].

Na geração aleatória sem restrição, os valores legais de uma variável possuem a mesma probabilidade de serem gerados. O tipo da variável determina automaticamente quais são valores legais. Os valores aleatórios são obtidos por meio de declarações `scv_smart_ptr < TYPE > var_name`. A classe `scv_smart_ptr` implementa os ponteiros originais de C++, porém com a capacidade de gerenciar automaticamente a memória alocada por objetos que deixam de ser usados pela aplicação. Por exemplo, valores aleatórios para uma variável de tipo inteiro não sinalizado, com 8 *bits*, podem ser gerados por meio do Código 4.

Algoritmo 4 : Exemplo de geração aleatória de valores.

```
#include <scv.h>

int sc_main (int argc, char* argv[]) {
  scv_smart_ptr< sc_uint<8> > data;

  cout <<"Value of data pre  randomize : " << endl;
  data->print();
  data->next(); // Randomize object data
  cout <<"Value of data post randomize : " << endl;
  data->print();
  return 0;
}
```

Na modalidade de geração aleatória com restrições, os valores legais que podem ser gerados aleatoriamente são restringidos por meio de expressões matemáticas. Estas ex-

pressões podem variar de uma simples restrição que define valores mínimo e máximo a uma expressão complexa envolvendo múltiplas variáveis. A classe que provê suporte para a criação de tais expressões é a classe `scv_constraint_base`. O engenheiro declara as expressões que serão usadas por meio da macro `SCV_CONSTRAINT`. O comando `SCV_CONSTRAINT(a() == 1 && b() == 1) || ( a() == 0))` é um exemplo de geração de valores aleatórios com restrição. Nesta expressão, os possíveis valores gerados são $(a = 0,\ b = 0)$, $(a = 0,\ b = 1)$, $(a = 1,\ b = 1)$. Como o par $(a = 1,\ b = 0)$ não satisfaz a expressão, ele não é considerado na geração aleatória. O Código 5 é um exemplo de restrição por meio de definição de valores mínimos e máximo para o valor da variável aleatória. O referido código restringe os valores para faixa de 10 a 100, mas excluindo os valores entre 21 e 49, e entre 61 e 80.

Algoritmo 5 : Exemplo de geração aleatória com restrição de valores.

```
//Restrição simples. Valores válidos para a são 10-20, 50-60, 90-100
struct faster : public scv_constraint_base {

   scv_smart_ptr <int> a;
   SCV_CONSTRAINT_CTOR(faster) {
      a->keep_only(10, 100);
      a->keep_out(21, 49);
      a->keep_out(61, 89);
   }
};
```

A modalidade de geração aleatória com peso é útil quando se deseja que certos valores sejam selecionados com mais freqüência que outros. Para tanto, o engenheiro especifica não somente um determinado valor que deseja, mas também o peso que tal valor tem em relação aos demais valores da distribuição. Uma distribuição com peso é criada por meio de dois passos básicos. Primeiro cria-se um objeto do tipo `scv_bag`. Um objeto deste tipo é uma coleção que permite a repetição de elementos, tal como um multi-conjunto. Em seguida, o objeto criado é povoado com os valores desejados. O peso de cada valor é relativo ao total de elementos dentro do objeto `sc_bag`. O Código 6 é um exemplo de geração de valores com peso na distribuição de valores. No referido código, 40 % dos valores gerados estarão no intervalo fechado de 0 a 1, e 60% dos valores estarão no intervalo fechado de 2 a 10.

Dentre as três modalidades de geração aleatória de valores, o nosso protótipo lida apenas

Algoritmo 6 : Exemplo de geração aleatória com peso na distribuição.

```
scv_smart_ptr<int> data;
scv_bag<pair<int, int> > dist;
dist.add(pair<int, int>(0,1), 40);
dist.add(pair<int, int>(2,10), 60);
data->set_mode(dist);
```

com geração de valores aleatórios com peso na distribuição. A justificativa para não lidar com a modalidade de geração sem restrições é a falta de controle por parte do engenheiro sobre os valores que são gerados. Como as únicas restrições são os próprios limites impostos pelo tipo da variável, o engenheiro não tem como configurar o gerador de estímulos para priorizar a satisfação dos critérios de cobertura. Além disso, o molde do componente Source gerado pela ferramenta eTBc, por exemplo, não considera esta modalidade de geração de valores aleatórios.

A metodologia VeriSC incentiva o uso da geração com restrições e da geração com peso na distribuição fornecendo código para tais modalidades no molde do componente Source. A ferramenta eTBC gera automaticamente códigos contendo as declarações das variáveis para que o engenheiro tenha apenas o trabalho de codificar o que é específico do bloco de projeto sendo verificado. Não lidamos com a modalidade de geração com restrições porque ela requer o desenvolvimento ou reuso de um solucionador de restrições que, por questões de complexidade, está fora do escopo deste trabalho. É importante observar que a técnica que propomos é válida para a modalidade de geração de valores aleatórios com restrição. Ela não está sendo considerada devido à dificuldade de se implementar um solucionador de restrições novo ou de se reusar um solucionador existente.

Do ponto de vista de implementação, constituir o conjunto C a partir de objetos `scv_bag` é bem mais simples que construí-lo a partir de restrições definidas por expressões matemáticas. A classe `scv_bag` contém os seguintes métodos:

- remove() : este método remove um ou todos os objetos com o mesmo valor do objeto passado como argumento.
- add() : este método adiciona ao multi-conjunto um valor passado com argumento.
- push() : este método adiciona ao multi-conjunto um valor na quantidade especificada

como argumento.

Da mesma maneira que tais métodos são usados para preencher os objetos do tipo `scv_bag` que serão utilizados para simular o bloco de projeto, eles podem ser usados para constituir o conjunto C que precisamos. Assim, os comandos contendo tais métodos são obtidos do componente Source e usados para constituir o conjunto C.

Ferramenta para Construção de Ambientes de Verificação na Fase de Integração

Na metodologia VeriSC, o ponto de partida para geração automática dos moldes do ambiente de verificação é a construção do arquivo contendo informações sobre os tipos de comunicação que serão utilizados no projeto. Definidos os tipos de comunicação, o engenheiro precisa especificar a hierarquia de blocos do projeto, as interfaces que ele contém e as FIFOs que serão utilizadas para ligar os blocos do ambiente de verificação. Exemplo de tal arquivo foi apresentado no Capítulo 2. De agora em diante, arquivos desta natureza serão referenciados por arquivos TLN, em alusão as suas extensões.

Os mesmos arquivos TLN já definidos para a criação dos moldes do ambiente de verificação são usados no processo de integração bloco a bloco. O engenheiro inicia a integração selecionando dois blocos que devem ser agrupados. Duas condições devem ser satisfeitas para realização da integração de cada bloco. Primeiro, os blocos selecionados devem ser vizinhos conforme definido no arquivo TLN, na parte que descreve o bloco macro. Segundo, cada bloco selecionado deve ter sido verificado isoladamente, isto é, o ambiente de verificação de cada um deles já deve ter sido construído. A primeira condição serve para garantir que estamos realizando uma integração sem problemas do ponto de vista de interfaces e tipos de dados envolvidos na comunicação. A segunda condição é necessária para permitir o reuso dos componentes do ambiente de verificação de cada bloco de projeto.

Satisfeitas as duas condições definidas no parágrafo anterior, o que nos falta é obter o ambiente de verificação para a integração sendo criada. Para tanto, precisamos de informações novas, que não são fornecidas pelos arquivos TLN originais. Estas informações são referências para os elementos do ambiente de verificação de cada bloco de projeto. O reuso de componentes no novo ambiente de verificação é obtido através de tais referências. Adicionalmente, alguns elementos precisam ser reconfigurados. Em relação ao bloco à esquerda da integração, a FIFO, que antes ligava o Modelo de Referência ao Checker, agora deve ser

configurada para ligar os dois modelos de referência da integração. Seguindo o mesmo princípio, os sinais produzidos pelo DUV à esquerda não devem ser mais ligados ao TMonitor, eles devem ser ligados ao DUV do lado direito.

Para cada bloco integrado, a implementação do ambiente de verificação não deve criar novos componentes, tais como geradores de estímulos e comparadores. A implementação deve ser baseada em referências aos componentes originais que foram desenvolvidos para a verificação de cada bloco de projeto isoladamente. Desta maneira, torna-se possível simular isoladamente os blocos da integração com novos cenários que foram obtidos pela aplicação da nossa técnica. Adicionalmente, o uso de referência também facilita a realização de testes de regressão [1] em relação à possibilidade de se existir múltiplas versões dos elementos do ambiente de verificação.

Para ilustrar o processo de construção dos ambientes de verificação na fase de integração com suporte ferramental, vamos considerar um sistema composto por três blocos conforme ilustrado na parte superior da Figura 3.10. Para integração Y∘X, precisamos do arquivo TLN, do arquivo de referências para o ambiente de verificação do bloco X (denotado por T_X na Figura 3.10) e do arquivo de referências para o ambiente de verificação do bloco Y (denotado por T_Y na Figura 3.10). Com estas informações, o ambiente de verificação para Y∘X pode ser gerado. A ferramenta que se encarrega de tal geração se chama eTBi (*easy TestBench integrator*)[2]. Realizada a integração de X e Y, o engenheiro pode realizar a integração do bloco Z, resultando no sistema Z∘Y∘X. Para tanto, ele precisa fornecer o mesmo arquivo TLN utilizado no passo anterior, o arquivo de referências para o ambiente de verificação do bloco Y∘X e o arquivo de referências para o ambiente de verificação do bloco Z.

A ferramenta eTBi é composta por três módulos básicos, conforme ilustrado na Figura 3.11. O primeiro bloco é o *parser* de arquivos TLN e o segundo é o *parser* de ambiente de verificação. Estes dois *parsers* transformam as informações textuais da TLN e dos ambientes de verificação em estruturas de dados adequadas ao processo de construção dos novos

[1] Teste de Regressão é uma prática comum no desenvolvimento de sistemas que consiste em aplicar testes à versão do sistema sendo alterada para garantir que não surgiram novos erros em partes do sistema que já foram testadas.

[2] O nome eTBi é uma alusão à ferramenta eTBc (*easy TestBench creator*), responsável pela geração automática dos moldes dos ambientes de verificação do bloco macro e cada bloco básico que compõe o bloco macro

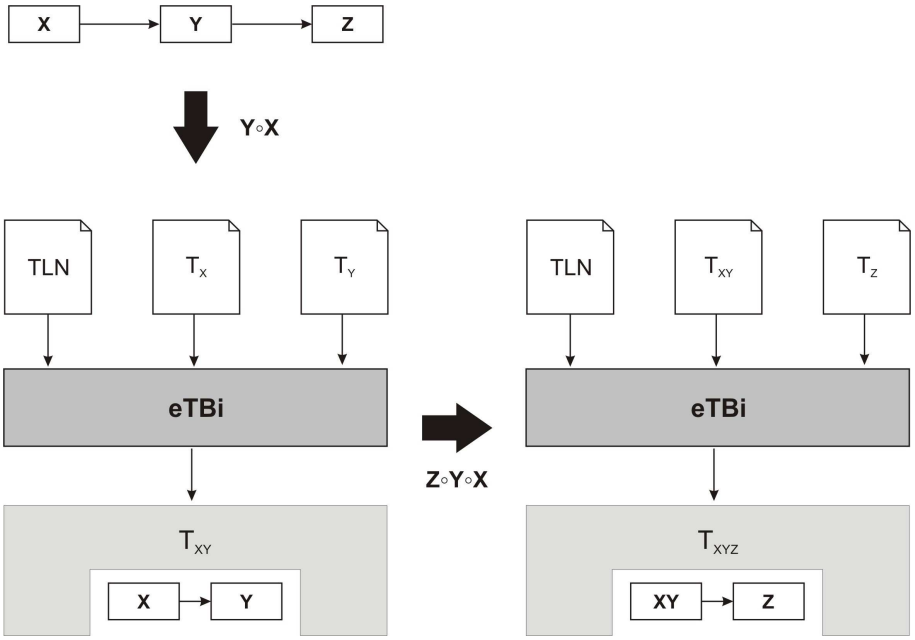

Figura 3.10: Exemplo de integração bloco a bloco com suporte ferramental na criação do ambiente de verificação da integração.

ambientes de verificação da integração. Esta construção é realizada pelo terceiro módulo, chamado de módulo integrador de ambientes de verificação na Figura 3.11.

3.5 Exemplo de Integração

Para ilustrar a aplicação da nossa abordagem, apresentamos um exemplo de integração. O sistema que vamos considerar é o DPCM (*Differential Pulse-Code Modulation*), apresentado no Capítulo 2. O DPCM recebe uma seqüência de amostras na entrada, faz a subtração entre a amostra corrente e a amostra anterior. Caso o resultado da subtração seja um valor fora de uma faixa pré-determinada, o resultado é saturado para um valor dentro da faixa e produzido como resultado de saída. No caso do DPCM como proposto na Figura 2.3, o sistema é formado por dois blocos: DIF e SAT. O bloco DIF realiza a subtração entre amostras e o bloco SAT verifica se o resultado da subtração está dentro da faixa pré-determinada.

Os códigos 7 e 8 são trechos do ambiente de verificação dos blocos SAT e DIF respectivamente. O Código 7 é parte do Source do bloco SAT e determina os estímulos utilizados na simulação do bloco. Na linha 03 do referido código, por exemplo, amostras não saturadas entre 63 (sessenta e três) e 127 (cento e vinte e sete) são produzidas para simular o bloco.

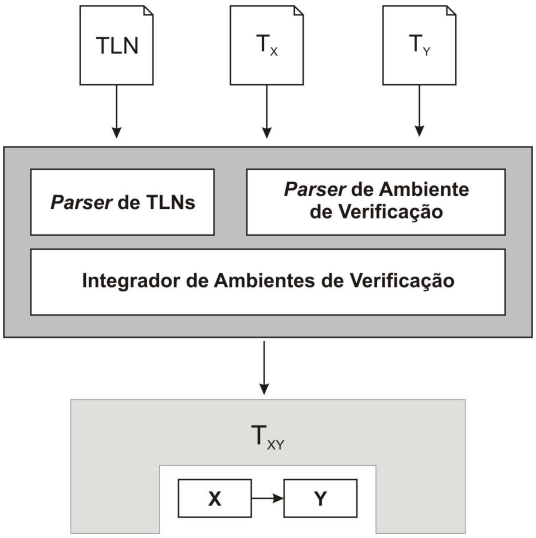

Figura 3.11: Arquitetura básica da ferramenta para geração de ambientes de verificação na fase de integração.

Para cada faixa existe um valor que determina a probabilidade dos valores serem utilizados na simulação. Na nossa abordagem, não estamos considerando a configuração de tais probabilidades. De fato, a distribuição de probabilidades dos estímulos tem influência no tempo de execução consumido para se alcançar as metas de cobertura do projeto. A configuração correta das probabilidades depende de conhecimentos sobre o funcionamento interno do bloco de projeto. Como o objetivo do nosso trabalho é identificar novos cenários de simulação que podem ser obtidos da integração e não melhorar o tempo de execução, vamos considerar apenas a faixa de valores sendo utilizada.

Algoritmo 7 : Fragmento de código para geração de estímulos do bloco de projeto SAT.

```
01   SCV_CONSTRAINT_CTOR(diff_in_constraint_class) {
02    unsat_sample_distrib.push(pair<int,int>(-128,-64), 100);
03    unsat_sample_distrib.push(pair<int,int>(63,127), 100);
04    unsat_sample_distrib.push(pair<int,int>(-20,20),100);
05    unsat_sample_distrib.push(pair<int,int>(-256,-200),1000);
06    unsat_sample_distrib.push(pair<int,int>(200,256),1000);
07    unsat_sample_distrib.push(pair<int,int>(-512,-500),1000);
08    unsat_sample_distrib.push(pair<int,int>(500,512),1000);
09  }
```

O princípio de considerar apenas a faixa de valores também é empregado em relação às especificações de cobertura dentro do componente TMonitor. A quantidade de vezes

que um valor deve ser medido dentro de uma declaração BVE_COVER_BUCKET não nos importa. No caso do TMonitor do bloco DIFF (vide Código 8), por exemplo, vamos considerar as faixas de valores não saturados que foram especificadas como critérios de cobertura.

Algoritmo 8 : Fragmento de código contendo especificação de cobertura para o bloco DIFF.

```
// Bucket
01   Cv_bckt_mnt_unsat_audio.begin();
02    BVE_COVER_BUCKET(Cv_bckt_mnt_unsat_audio, tr_ptr->unsat_sample>=-128 &&
                                        tr_ptr->unsat_sample<=-64,100);
03    BVE_COVER_BUCKET(Cv_bckt_mnt_unsat_audio, tr_ptr->unsat_sample >= 64 &&
                                        tr_ptr->unsat_sample <= 127,100);
04    BVE_COVER_BUCKET(Cv_bckt_mnt_unsat_audio, tr_ptr->unsat_sample >= -1024  &&
                                        tr_ptr->unsat_sample <= -1000,100);
05    BVE_COVER_BUCKET(Cv_bckt_mnt_unsat_audio, tr_ptr->unsat_sample >= 1000  &&
                                        tr_ptr->unsat_sample <= 1023,100);
06   Cv_bckt_mnt_unsat_audio.end();

// Ignore
07   Cv_ignr_mnt_unsat_audio.begin();
08    BVE_COVER_IGNORE(Cv_ignr_mnt_unsat_audio, tr_ptr->unsat_sample <= 10 &&
                                        tr_ptr->unsat_sample >= -10);
09   Cv_ignr_mnt_unsat_audio.end();

// Illegal
10   Cv_illg_mnt_unsat_audio.begin();
11    BVE_COVER_ILLEGAL(Cv_illg_mnt_unsat_audio, tr_ptr->unsat_sample > 1023);
12    BVE_COVER_ILLEGAL(Cv_illg_mnt_unsat_audio, tr_ptr->unsat_sample < -1024);
13   Cv_illg_mnt_unsat_audio.end();
```

Para facilitar a visualização da aplicação da nossa abordagem, na Figura 3.12 apresentamos cada conjunto relevante que é obtido para o DPCM. Para cada conjunto é usado uma reta na horizontal representando o eixo dos números inteiros. Cada segmento deste eixo destacado por uma linha mais espessa representa que os referidos valores fazem parte do conjunto. Um círculo fechado representa que o referido valor faz parte do conjunto e um círculo vazio denota que o referido valor não faz parte do conjunto. A união dos valores representados por cada segmento mais espesso é o conjunto de valores propriamente.

Em relação à Figura 3.12, no primeiro eixo temos os valores do conjunto A (vide Figura 3.7), obtido pela união das especificações de BVE_COVER_ILLEGAL e BVE_COVER_IGNORE do bloco DIFF. Ainda em relação ao bloco DIFF, o eixo seguinte

contém os valores que foram especificados como cobertura funcional, ou seja, este eixo contém valores do conjunto B. No terceiro eixo, temos o conjunto de valores que são utilizados na simulação do bloco SAT, isto é, temos o conjunto C. Do quarto eixo em diante, temos os conjuntos que são obtidos pela aplicação da nossa técnica. O quarto eixo é o conjunto E. Ele contém os valores que precisam ser notificados ao engenheiro para alertá-lo de que é possível que a cobertura funcional não alcance 100 % na verificação funcional da integração dos dois blocos. Os referidos valores foram utilizados na simulação do bloco SAT, mas foram explicitamente ignorados na saída do bloco DIFF. Assim, pode ser o caso que tais valores não sejam produzidos na simulação da integração. O quinto eixo é o conjunto F. Ele representa os novos valores que devem ser especificados como critérios de cobertura do bloco DIFF. O Código 9 é a implementação deste conjunto. A multiplicidade de cada item BVE_COVER_BUCKET foi obtida manualmente. O último eixo é o conjunto G. Ele representa os valores que devem ser utilizados na simulação do bloco SAT. O Código 11 é uma implementação deste conjunto na forma de declarações do tipo BVE_COVER_BUCKET. Este código deve compor o componente TDriver do bloco de projeto SAT. O Código 10 é exemplo de implementação de parte do Source que gera os valores do conjunto G e que são capazes de satisfazer às coberturas do Código 11.

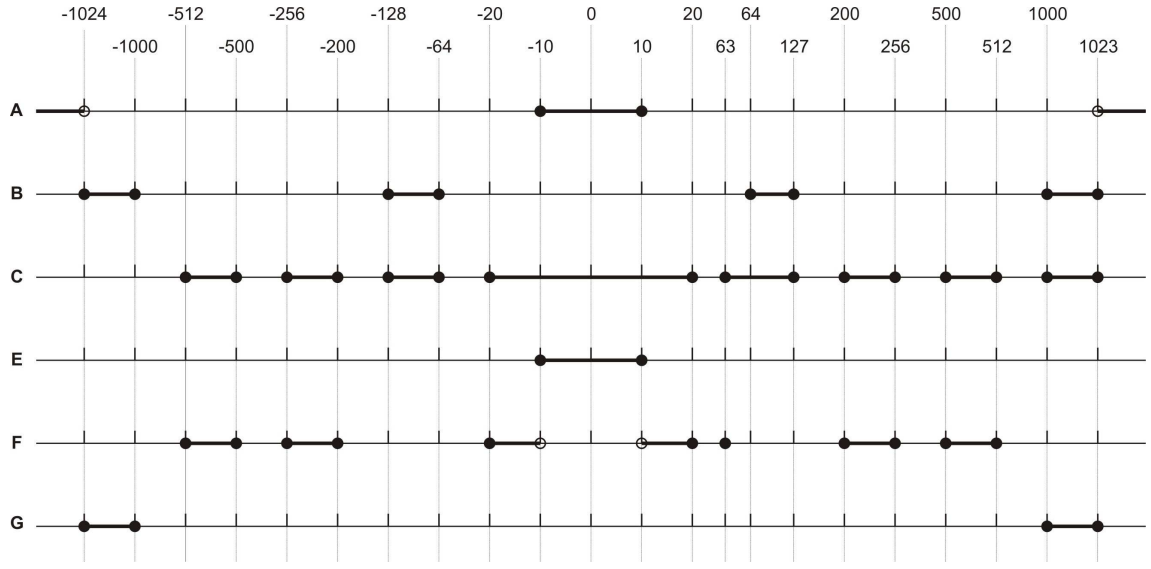

Figura 3.12: Representação dos conjuntos A, B, C, E, F e G para uma especificação do DPCM.

Algoritmo 9 : Fragmento de código contendo especificação de cobertura para o bloco de projeto DIFF.

```
// Bucket
Cv_bckt_mnt_piracema01.begin();
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample>=-512 &&
                                          tr_ptr->unsat_sample <= -500,100);
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample >= -256 &&
                                          tr_ptr->unsat_sample <= -200,100);
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample >= -20  &&
                                          tr_ptr->unsat_sample < -10,100);
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample > 10  &&
                                          tr_ptr->unsat_sample <= 20,100);
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample = 63,100);
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample >= 200 &&
                                          tr_ptr->unsat_sample <= 256,100);
 BVE_COVER_BUCKET(Cv_bckt_mnt_piracema01, tr_ptr->unsat_sample >= 500 &&
                                          tr_ptr->unsat_sample <= 512,100);
Cv_bckt_mnt_piracema01.end();
```

Algoritmo 10 : Fragmento de código para a geração de novos estímulos para simulação do bloco SAT.

```
SCV_CONSTRAINT_CTOR(diff_in_constraint_class) {
 unsat_sample_distrib.push(pair<int,int>(-1024,-1000),100); // by Piracema
 unsat_sample_distrib.push(pair<int,int>(1000,1023),100); // by Piracema
}
```

Algoritmo 11 : Fragmento de código contendo cobertura funcional para o bloco SAT.

```
// Bucket
Cv_bckt_drv_piracema01.begin();
 BVE_COVER_BUCKET(Cv_bckt_drv_piracema01, tr_ptr->unsat_sample >= -1024 &&
                                          tr_ptr->unsat_sample <= -1000,100);
 BVE_COVER_BUCKET(Cv_bckt_drv_piracema01, tr_ptr->unsat_sample >= 1000 &&
                                          tr_ptr->unsat_sample <= 1023,100);
Cv_bckt_drv_piracema01.end();
```

3.6 Problemas com Reuso na Integração

Dependendo da topologia do diagrama de blocos, o processo de geração do ambiente de verificação da integração pode gerar diversas versões dos componentes Source e Checker. Seja um sistema qualquer com diagrama de blocos conforme ilustrado no item *a* da Figura 3.13, a integração Y∘Z, por exemplo, pode levar a construção de um novo componente Source para estimular o bloco Z, que não pode ser usado na simulação do bloco Z isoladamente, isto é, o novo Source serve somente para integração.

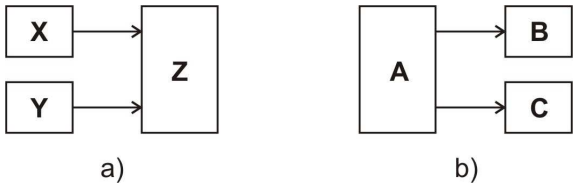

Figura 3.13: Exemplo de topologia com três blocos.

Conforme ilustrado na parte *b)* da Figura 3.14, o componente SourceZ foi originalmente implementado para gerar estímulos através de duas interfaces distintas: uma se refere aos estímulos que são produzidos pelo bloco X e a outra se refere aos blocos que são produzidos pelo bloco Y. Quando ocorre a integração, o SourceZ não precisa mais gerar os estímulos referentes ao bloco Y, pois estes estímulos serão fornecidos pelo bloco Y de fato (vide Figura 3.14, item *c)*). Da forma como os geradores de estímulos são construídos em VeriSC, não é possível desabilitar por meio de *programação* a geração dos estímulos referentes ao bloco Y. A geração de tais estímulos é desabilitada por meio de *edição*, através da qual o engenheiro deve remover os trechos de código referente à geração dos estímulos. Após esta integração, quando for o caso de executar novamente a verificação do bloco Z isoladamente, os trechos de código que foram removidos do componente SourceZ precisam ser inseridos novamente para que estímulos sejam gerados para as duas interfaces. A existência de múltiplas versões do componente Checker é indesejável porque exige o custo adicional de manter a consistência entre os estímulos que são gerados para o bloco isoladamente e os que são gerados para a integração.

Este trabalho de inserir e remover código não é necessário para o componente SourceY da integração, pois ele pode ser reusado integralmente em seu ambiente de verificação original e no ambiente de verificação da integração(vide Figura 3.14, item *a)*).

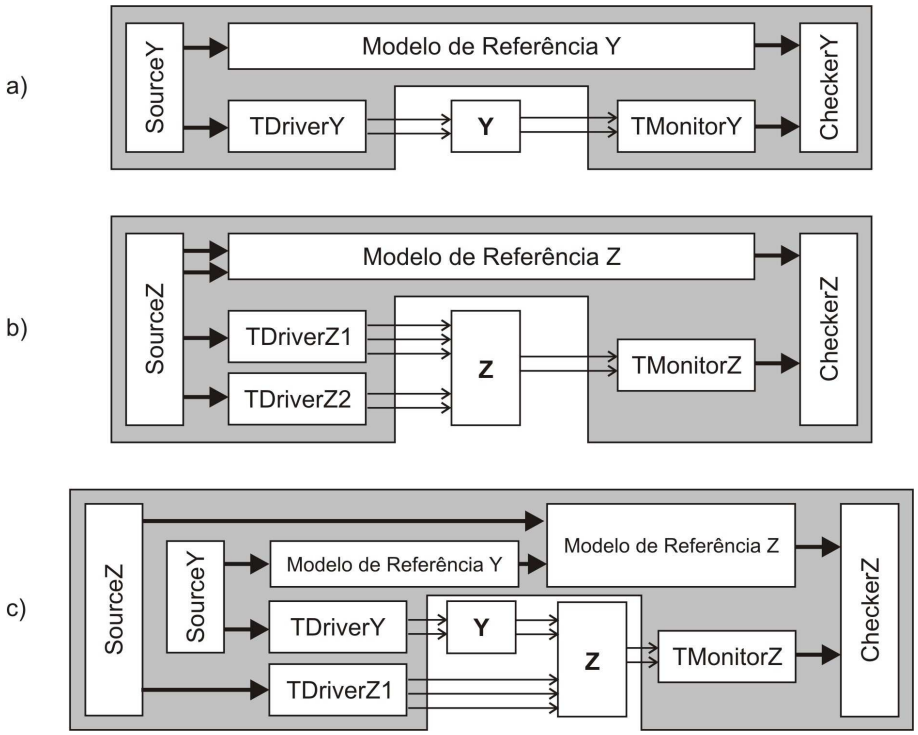

Figura 3.14: Exemplos de ambiente de verificação para o sistema da Figura 3.13, item *a*.

Um sistema com diagrama de blocos conforme ilustrado no item *b* da Figura 3.13 também apresenta problemas no reuso dos componentes do ambiente de verificação. A integração B∘A, por exemplo, pode levar à construção de um novo componente Checker para o bloco A, que não pode ser usado na simulação do bloco A isoladamente, isto é, o novo Checker serve somente para integração.

Conforme ilustrado na parte *a)* da Figura 3.15, o componente Checker do bloco A recebe valores de duas interfaces, uma referente aos dados que são produzidos para o bloco B e outra referente aos dados que são repassados para o bloco C. Quando ocorre a integração do bloco A com o bloco B, conforme ilustrado no item *c* da Figura 3.15, o componente Checker do bloco A deixa de receber dados da interface referente à comunicação com o bloco B, pois eles são repassados para o bloco B de fato. Conseqüentemente, o componente Checker da integração passa a receber apenas dados que seriam repassados para o bloco C. Isto leva a uma edição do componente Checker original de A para que ele funcione conforme ilustrado na Figura 3.15, item *c*. Assim, duas versões do componente Checker são necessárias, uma para a verificação do bloco A isoladamente e outra para verificação da integração B∘A.

A lógica interna do componente Checker pode ser considerada simples em relação aos

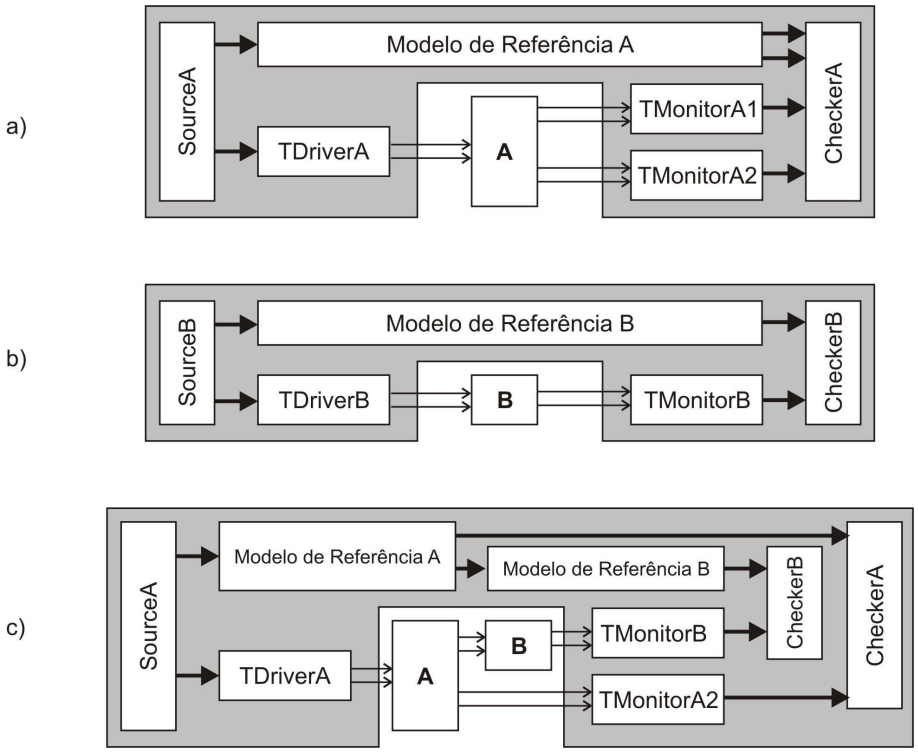

Figura 3.15: Exemplos de ambiente de verificação para o sistema da Figura 3.13, item *b*.

demais componentes do ambiente de verificação, tais como geradores de estímulos e monitores. O papel de um Checker é apenas comparar os resultados produzidos pelo Modelo de Referência e pelo DUV. Assim, o melhoramento na especificação de geradores de estímulos e de critérios de cobertura não afeta diretamente o bloco Checker. No entanto, a existência de múltiplas versões deste componente requer, por parte do engenheiro, o trabalho adicional de configurar o ambiente de verificação para considerar os diversos componentes do tipo Checker que são produzidos na medida em que blocos de projeto são integrados. Este trabalho adicional poderia ser suprimido caso fosse possível reusar o componente Checker na verificação isolada do bloco de projeto ou em alguma situação de integração. Em VeriSC, contudo, isso não é possível.

3.6.1 Refatoração do Ambiente de Verificação de VeriSC

Diante dos problemas com reuso dos componentes Source e Checker apresentados na seção anterior, é possível propor alterações na arquitetura do ambiente de verificação funcional de VeriSC para que tais problemas não mais ocorram. Estas alterações não modificam o

comportamento funcional de VeriSC, mas podem prover a flexibilidade e a simplicidade necessárias para que os componentes mencionados sejam reusados de forma transparente em relação ao contexto da verificação sendo realizada: verificação do bloco isolado ou verificação do bloco na fase de integração.

Um único componente Source pode não servir ao mesmo tempo para a verificação isolada do bloco e para a verificação na fase da integração se este Source tem como responsabilidade a geração de estímulos para mais de uma interface. Em outros termos, este problema ocorre porque, em VeriSC, um único componente Source é projetado para simular toda a parte do ambiente que produz estímulos para o bloco de projeto sendo verificado. Não importa se o bloco recebe dados de um, dois ou três blocos vizinhos, um único componente Source vai ser criado para simular a produção de estímulos pela sua vizinhança. Na medida em que ocorre a integração com os blocos vizinhos, este componente Source precisa ser alterado para que ele não gere mais os estímulos que são fornecidos pelo bloco vizinho que está sendo integrado.

Em VeriSC, diferentemente do componente Source, existe um componente TDriver para cada interface de entrada de dados. Esta característica faz com que componentes do tipo TDriver sejam reutilizados na verificação da integração tais como eles foram utilizados na verificação dos blocos isoladamente. Isto porque esses componentes apresentam um grau de coesão suficiente para desacoplá-los do contexto em que são utilizados. Este mesmo grau de coesão pode ser obtido para os geradores de estímulos fazendo-se com que para cada componente TDriver exista um bloco gerador de estímulos. Assim, em vez de possuir único Source para simular a produção de estímulos de todos os blocos vizinhos, o ambiente de verificação vai possuir um bloco Source para cada interface de entrada, isto é, cada vizinho tem a sua geração de estímulos simulada por um componente Source específico. Na medida em que os blocos vizinhos são integrados, os respectivos componentes do tipo Source são eliminados do ambiente de verificação. E como a eliminação de um componente gerador de estímulos não influência os demais geradores de estímulos, esta estratégia vai permitir que componentes do tipo Source sejam reusados de forma transparente em relação ao contexto da verificação: verificação do bloco isolado ou verificação do bloco na fase de integração.

A Figura 3.16 é uma ilustração do ambiente de verificação do bloco de projeto Z da Figura 3.13, item *a)*, com um componente do tipo Source para cada interface de entrada. É importante observar na Figura 3.16 que os componentes SourceZ1 e SourceZ2 não estão

acoplados. A implementação de um não depende do outro e vice-versa. Assim, na integração de algum bloco vizinho na entrada do bloco de projeto Z, o respectivo bloco Source pode ser removido sem a necessidade de editar o outro bloco Source que será integrado em seguida.

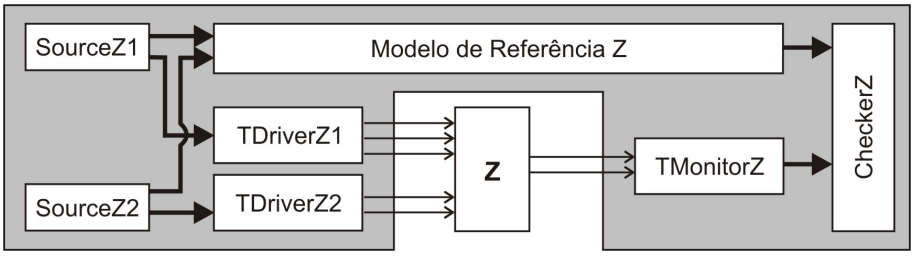

Figura 3.16: Ambiente de verificação para o bloco de projeto Z da Figura 3.13.

Para resolver o problema com o reuso dos componentes do tipo Checker, podemos seguir o mesmo raciocínio usado para resolver o problema com o reuso de componentes do tipo Source. Assim, para cada interface de saída de um bloco de projeto, vai existir um componente do tipo Checker. Esta estratégia também é capaz de tornar os componentes do tipo Checker coesos a ponto de que o seu reuso seja transparente em relação ao contexto da verificação. A Figura 3.6.1 é uma ilustração do ambiente de verificação do bloco de projeto A da Figura 3.13, item *b)*, com um componente do tipo Checker para cada interface de saída. Observe que para cada interface de saída existe um TMonitor e um Checker.

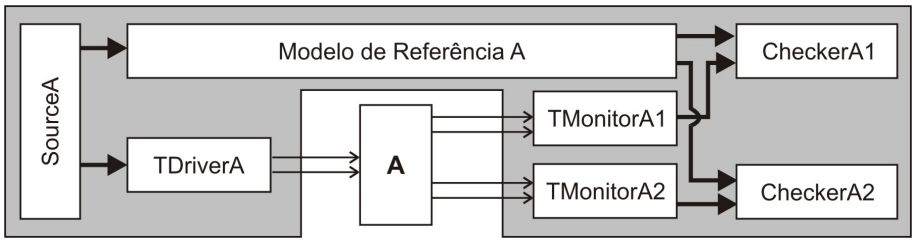

Figura 3.17: Ambiente de verificação para o bloco de projeto A da Figura 3.13.

3.7 Aplicação da Técnica em Outras Metodologias de Verificação Funcional

Esta seção tem como objetivo caracterizar a abrangência da abordagem proposta em relação à metodologia de verificação funcional subjacente. Para esta caracterização será considerada a

arquitetura do ambiente de verificação de três metodologias de verificação funcional: BVM, OVM e VMM. Estas metodologias foram previamente analisadas na Seção 2.5.

3.7.1 BVM

BVM (*Brazil-IP Verification Methodology*) é uma reformulação da metodologia VeriSC. As motivações para tal reformulação são decorrentes das dificuldades de usar SystemC para verificação funcional. SystemC, por exemplo, não possui uma biblioteca nativa para especificação de cobertura funcional. Assim, a cobertura funcional de VeriSC só pode ser usada em projetos desenvolvidos em VeriSC. Em resposta a tais dificuldades intrínsecas de SystemC, uma nova linguagem foi desenvolvida como padrão: SystemVerilog [11]. Em relação à SystemC, um dos maiores benefícios de SystemVerilog é que ela permite a criação de ambientes de verificação confiáveis, repetíveis e em uma sintaxe consistente que podem ser usados em vários projetos.

A arquitetura do ambiente de verificação de BVM está ilustrado na Figura 3.18. A principal novidade em relação à metodologia VeriSC é a existência do componente Actor. Tal componente é responsável por controlar e monitorar os sinais do protocolo de comunicação entre o DUV e o ambiente de verificação. Conforme ilustrado na Figura 3.18, ele é inserido entre o DUV e o Monitor. Do ponto de vista prático, ele viabiliza a monitoração de sinais entre DUVs que foram integrados. Isto é possível porque ele permite que os sinais do protocolo de comunicação sejam repassados para um Monitor e em seguida para o Checker. Em VeriSC isto não é possível porque a única forma de monitorar os sinais que saem de um DUV por uma determinada interface é por meio de um único componente TMonitor. Mas quando ocorre a integração com um DUV vizinho, este componente é descartado.

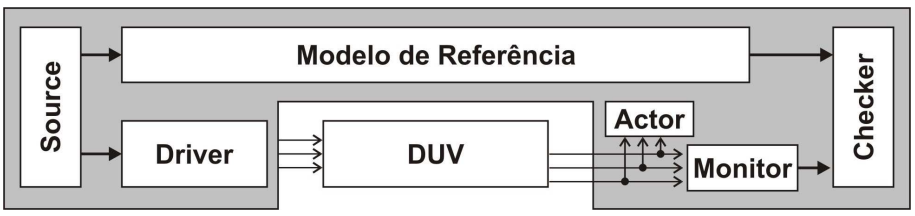

Figura 3.18: Ambiente de verificação da metodologia BVM.

A existência do componente Actor torna possível que o Monitor deixe de ser descartado

na integração. Se ele não é descartado, os critérios de cobertura especificados em tal componente existirão também para integração. Isto significa que o engenheiro pode optar por não realizar a análise dos componentes do ambiente de verificação que são *opcionalmente* descartáveis , já que é possível construir o ambiente de verificação para os blocos integrados de modo que eles permaneçam na simulação da integração. Mas deve ser observado que, ao optar por não realizar tal análise, o engenheiro está abrindo mão de que o ambiente de verificação de cada bloco seja melhorado em relação a seus blocos vizinhos. Assim, na medida em que um bloco é empregado em projetos diferentes, a especificação do seu ambiente de verificação se mantém e perde a oportunidade de ser melhorado a partir da especificação de seus blocos vizinhos.

SystemVerilog tem suporte à especificação dos três tipos de critério de cobertura considerados neste trabalho. Assim, se o engenheiro optar por confrontar a especificação do ambiente de verificação de cada bloco com a vizinhança, os procedimentos que devem ser seguidos são os mesmos definidos para VeriSC.

3.7.2 OVM

A metodologia OVM oferece uma biblioteca para verificação funcional que combina geração de testes automáticos, ambientes autoverificáveis e métricas de cobertura para reduzir o tempo consumido na verificação de um projeto.

O ambiente de verificação em OVM é composto de componentes de verificação reusáveis denominados de OVC (*OVM Verification Component*). Ele foi projetado de modo a ser configurável para ser utilizado em diversos contextos, tais como protocolo de interface, submódulo de um projeto, ou para um sistema inteiro. Conforme ilustrado na Figura 3.19, os componentes que constituem o ambiente de verificação de OVM assemelham-se aos componentes utilizados na metodologia VeriSC tanto em nome quanto em organização.

O componente *Stimulus* tem a mesma função do Source em VeriSC. Ele opera como um gerador de estímulos que controla os itens de dados fornecidos para o Driver. O componente *Test Controller* é responsável por coordenar a simulação. Tipicamente, os *controllers* recebem informações sobre os *scoreboards* e cobertura e mandam informações para outros componentes do ambiente de verificação. Como os critérios de cobertura em OVM também podem ser especificados nos componentes *Monitor*, a verificação funcional na fase de

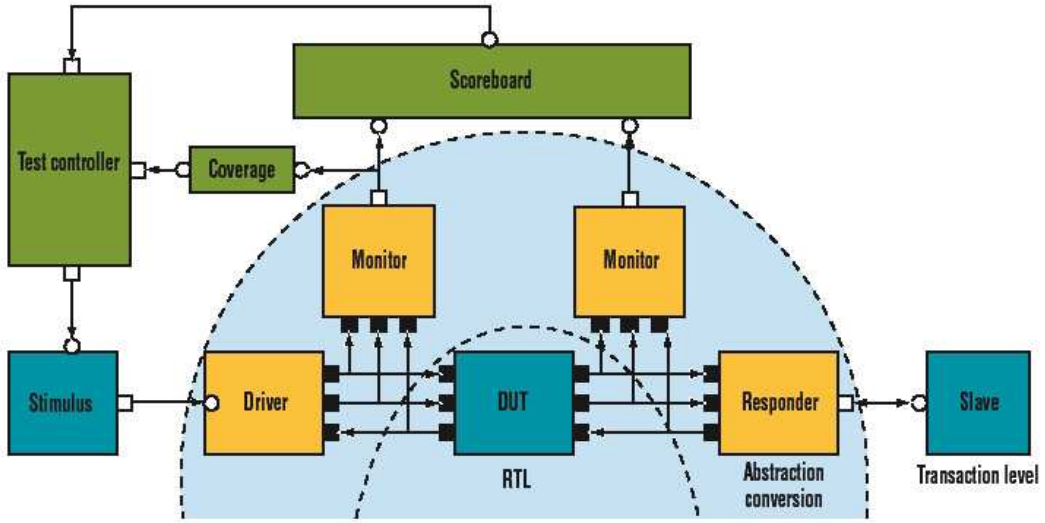

Figura 3.19: Ambiente de verificação na metodologia OVM.

integração em OVM pode se valer da abordagem proposta neste documento.

O componente *Responder* executa papel semelhante ao executado pelo componente *Agent* de BVM. Isto significa que em OVM também é possível construir um ambiente de verificação para a integração sem que os critérios de cobertura de cada bloco integrado sejam descartados. Mas novamente deve ser observado que, ao não aplicar abordagem proposta neste trabalho, o engenheiro está abrindo mão de que a especificação de cobertura funcional de cada bloco seja confrontada com a especificação de seus blocos vizinhos.

3.7.3 VMM

A metodologia VMM segue uma arquitetura em camadas para organizar ambiente de verificação. Esta arquitetura possui cinco camadas que estão distribuídas conforme indicado na Figura 3.20. A camada de sinal (*signal layer*) tem como propósito básico conectar o ambiente de verificação com o projeto em RTL (DUV). A camada de comandos (*command layer*) contém *drivers*, monitores e asserções de mais baixo nível. Para abstrair detalhes da camada de sinal, ela provê uma interface no nível de transação para a camada superior. A camada funcional (*functional layer*) contém *drivers* e monitores de mais alto nível assim como a estrutura que torna o ambiente de verificação autoverificável. A camada de cenários (*scenario layer*) possui geradores para produzir as seqüências de transações que são submetidas à camada funcional. É nesta camada que o engenheiro configura a geração aleatória de

estímulos em VMM. Os testes propriamente ficam contidos na camada de testes (*test layer*). Esta camada pode interagir com todas as camadas.

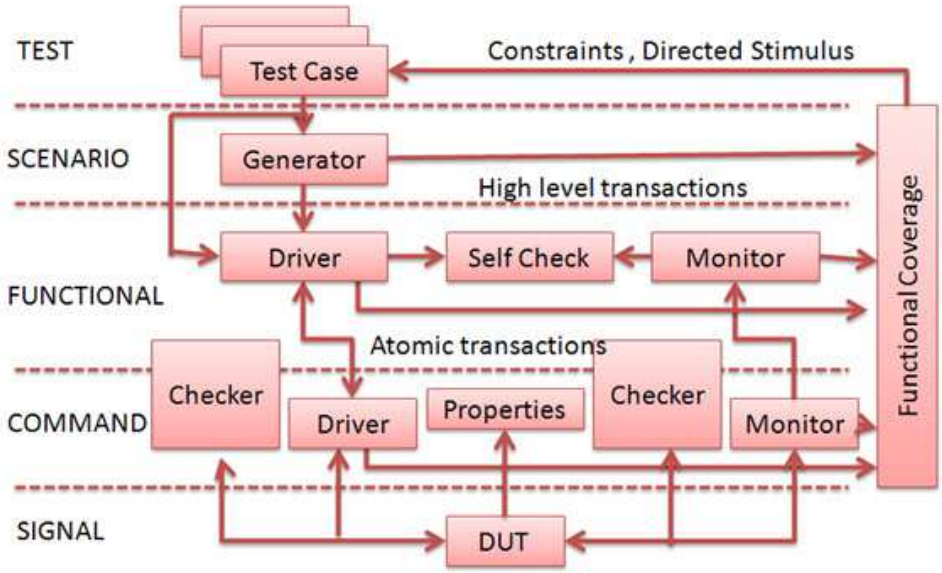

Figura 3.20: Ambiente de verificação na metodologia VMM.

A organização em camadas do ambiente de verificação não inviabiliza a aplicação da abordagem proposta neste documento. Por ser implementada em SystemVerilog, VMM suporta as modalidades de cobertura funcional consideradas em VeriSC. Em relação à metodologia VeriSC, a novidade que deve ser observada é a distinção entre *drivers* e monitores das camadas de comandos e funcional. Esta diferenciação permite que a cobertura seja medida nos níveis de sinal e de transação. Em VeriSC, não existe tal diferenciação e a análise de componentes descartados proposta neste trabalho é realizada sobre critérios de cobertura definidos no nível de transações. Assim, no caso de VMM, a aplicação da abordagem proposta deve ser feita na camada funcional.

3.8 Considerações Finais do Capítulo

Este capítulo apresentou uma forma sistemática de se proceder a verificação funcional na fase de integração de blocos de projetos de circuitos digitais. A partir da análise de compo-

nentes do ambiente de verificação que são descartados na verificação funcional da integração, novos critérios de cobertura podem ser calculados para cada bloco da integração. Dados dois blocos de projeto, A e B, tais que os valores produzidos na saída A são utilizados para alimentar o bloco B, o cálculo é realizado analisando-se as informações que estão registradas nos componentes TMonitor de A e no Source de B. No entanto, nada impede que os novos critérios de cobertura sejam obtidos pelo cruzamento de informações que estão nos componentes TMonitor de A e nos componentes TDriver de B, pois os critérios de cobertura no TDriver de B constituem um subconjunto do conjunto de valores gerados pelo Source de B. A diferença é que, ao considerar as informações que estão no Source de B, a análise de novos critérios de cobertura se baseia em valores de estímulos que *podem* ocorrer na simulação do bloco B. Ao considerar o TDriver de B, a análise de novos cenários se baseia em valores de estímulos que *devem* ocorrer na simulação de B.

A análise de componentes do ambiente de verificação que são descartados na verificação funcional da integração serve também para avaliar se algum critério de cobertura da integração tem a possibilidade de não ser satisfeito em função do descarte de componentes. Além disso, observou-se que a organização do ambiente de verificação pode dificultar o reuso dos componentes de verificação na fase integração. A existência de um gerador de estímulos para cada interface de entrada e um verificador para cada interface de saída permite que os componentes do ambiente de verificação sejam reusados na integração tal como foram usados na verificação isolada de cada bloco.

Capítulo 4

Resultados Experimentais

Este capítulo é dedicado à apresentação de resultados experimentais obtidos mediante a aplicação da técnica proposta. Inicialmente, é apresentada uma discussão sobre a seleção de projetos de circuitos digitais utilizados para validação do trabalho proposto. Esta discussão envolve a definição de critérios técnicos para seleção de projetos de circuitos digitais, apresentação do universo de projetos candidatos a serem utilizados e seleção de tais projetos de acordo com os critérios definidos.

Outro ponto abordado neste capítulo é a seleção de métricas adequadas para medir os benefícios advindos do trabalho proposto. A seleção de métricas se baseou nos objetivos do trabalho proposto, apresentados no Capítulo 1. A idéia de derivar métricas de projetos a partir de objetivos previamente declarados é utilizada na técnica conhecida como GQM (*Goal Question Metric*)[17]. Em seguida, uma visão geral de cada projeto selecionado é apresentada. O objetivo principal de apresentar esta visão geral é mostrar a complexidade de cada sistema em relação a sua hierarquia de blocos de projetos. Nas demais seções, cada tópico discutido no Capítulo 3 é abordado do ponto de vista prático. A apresentação do código referente à integração dos blocos de projeto é demasiado extensa para ser realizada neste documento. Assim, para resumir a apresentação das integrações de blocos de projetos e ainda fornecer uma noção da complexidade da atividade realizada, a representação gráfica do ambiente de verificação será utilizada.

4.1 Seleção de Métricas

A avaliação dos benefícios advindos do trabalho proposto depende de uma seleção criteriosa de métricas de projeto. É muito importante que as métricas utilizadas quantifiquem de fato atributos associados aos objetivos do trabalho proposto. Além disso, as métricas devem ser confiáveis no sentido de produzir os mesmos resultados quando as mesmas condições dos experimentos são mantidas. Elas também devem ser fáceis de computar e de interpretar.

A seleção de métricas para esta parte do trabalho é inspirada na abordagem GQM. Em GQM, primeiro estabelecem-se os principais objetivos da medição para o projeto. Exemplos de objetivos são: reduzir a quantidade de defeitos, aumentar produtividade, etc. A partir dos objetivos, são geradas as perguntas cujas respostas dirão se os objetivos foram ou não alcançados. Exemplos de pergunta são: qual a taxa de defeito atual? Qual a taxa de defeito após a implantação do novo processo? A partir das perguntas, são definidas as métricas, tais como número de defeitos e percentual de cobertura funcional atingida.

Conforme discutido no Capítulo 1, o objetivo deste trabalho é sistematizar a verificação funcional na fase de integração de blocos de projeto de circuitos digitais para que novos cenários emergentes da interação entre blocos sejam explorados. Esta sistematização deve permitir o reuso dos componentes de verificação, o melhoramento da especificação de critérios de cobertura dos blocos e redução do tempo na verificação funcional da integração.

A partir deste objetivo, as seguintes questões podem ser elaboradas:

1. **Questão 1:** para cada bloco da integração, qual a quantidade de cenários exercitados na verificação funcional?

2. **Questão 2:** para cada bloco da integração, qual a quantidade de erros detectados na verificação funcional?

3. **Questão 3:** quanto do ambiente de verificação de cada bloco da integração precisa ser editado para produzir o ambiente de verificação funcional da integração?

4. **Questão 4:** quanto tempo um engenheiro leva para realizar uma integração?

As questões 1 e 2 dizem respeito ao impacto do trabalho em relação à qualidade da verificação funcional. No Capítulo 3 provou-se que é possível revelar novos critérios de cobertura

funcional que não foram considerados na simulação dos blocos individualmente. Embora esta possibilidade tenha sido provada no referido capítulo, um ponto importante que precisa ser investigado na prática é se os novos critérios de cobertura funcional levam de fato a exploração de novos cenários. Para tal investigação, é necessário empregar outra métrica, além da cobertura funcional, que represente a simulação de novos cenários. Assim, em relação à Questão 1, as métricas de cobertura estrutural, tais como número de linhas exercitadas e número de caminhos de execução exercitados, podem ser utilizadas. Por exemplo, se o número de linhas ou de caminhos exercitados aumentar, significa que novos cenários de simulação passaram a ser considerados.

Em relação à Questão 2, melhorar a verificação funcional de um projeto pode levar a detecção de erros que até então não tinham sido detectados. Na Engenharia de Software, a qualidade dos testes em relação à detecção de comportamentos não esperados pode ser avaliada por meio da *análise de mutantes*. Na análise de mutantes, diversas variações do código original, chamadas de mutantes, são geradas de forma sistemática por meio de ferramentas. Estas variações são mímicas de *erros* de codificação que podem ser cometidos pelo programador. Em seguida, os mutantes são submetidos aos casos de testes definidos para o sistema. Idealmente, na realização de tais testes, todos os mutantes que não são funcionalmente equivalentes ao código original devem ser apontados como programas que não estão de acordo com a especificação. Quando algum mutante não equivalente ao programa original não é assim notado, significa que os casos de testes não foram adequadamente especificados. Conseqüentemente, os casos de testes precisam ser melhorados para detectar tais mutações.

Embora tenha sido desenvolvida originalmente para testes de software, a análise de mutantes pode fornecer uma métrica objetiva para se avaliar a qualidade da verificação funcional de circuitos digitais. Os trabalhos de Vado [83] e Serrestou [77], por exemplo, empregam análise de mutantes para o melhoramento dos vetores de teste utilizados na validação de circuitos digitais. Portanto, em relação à Questão 2 a métrica a ser utilizada será o número de mutantes mortos pelo ambiente de verificação funcional.

As questões 3 e 4 estão relacionadas à produtividade na fase de integração dos blocos de projeto. Em relação à questão 3, a verificação funcional de cada bloco isoladamente requer a produção de um ambiente de verificação funcional. Reusar os componentes do ambiente de verificação funcional de cada bloco isolado na fase de integração pode reduzir o trabalho dos

engenheiros. A métrica que pode indicar a redução deste trabalho é o percentual de linhas de código dos componentes de verificação de cada bloco da integração que precisam ser editadas para se produzir o ambiente de verificação da integração. A Questão 4 diz respeito à forma mais básica de se analisar a produtividade do engenheiro na fase de integração, que é medir o tempo gasto com tal atividade.

4.2 Seleção de Projetos

4.2.1 Critérios de Seleção

A seleção de projetos de circuitos digitais para realização de experimentos foi baseada em um conjunto de características que o projeto deve possuir para ser utilizado nesta fase do trabalho. Uma característica importante para aplicar a estratégia de integração proposta neste trabalho é que a verificação funcional do projeto seja dirigida por coberturas. Dado que o objeto de estudo principal deste trabalho é o melhoramento da cobertura funcional, é imprescindível que o ambiente de verificação do projeto contemple a especificação de critérios de cobertura funcional.

Outra característica essencial é que a verificação funcional seja do tipo *Black Box*. Se ela não for do tipo *Black Box*, existe a possibilidade de que critérios de cobertura estejam espalhados na implementação do modelo de referência. Caso os critérios de cobertura estejam espalhados no modelo de referência, não é possível aplicar a técnica proposta neste trabalho. Portanto, é fundamental que os critérios de cobertura estejam concentrados nas interfaces de entrada e saída dos blocos, tal como ocorrem em metodologias *Black Box*.

Na seção anterior, ficaram definidas as métricas que serão empregadas para avaliar os benefícios do trabalho proposto. Estas métricas também influenciam a seleção de projetos, pois se o projeto tiver sido desenvolvido sobre um conjunto de ferramentas sem suporte para realizar as medições, não faz sentido usá-lo nesta parte do trabalho. Portanto, o projeto precisa também de suporte ferramental para a realização de medições. Não é esperado que cada projeto tenha suporte a todas as métricas selecionadas anteriormente. Para ser selecionado, o projeto deve suportar pelo menos uma das medições mencionadas.

A última característica não diz respeito à aplicação da técnica propriamente. Esta é uma

característica que diz respeito à possibilidade de realizar experimentos. Para realizar os experimentos, o código do ambiente de verificação deve estar disponível. Sem esta disponibilidade, não há como realizar integrações, medições e outras análises.

4.2.2 Projetos Candidatos

Tendo em vista a complexidade e a organização do diagrama de blocos, o conjunto de projetos candidatos a esta parte do trabalho foi formado pelos sistemas listados a seguir:

1. DPCM (*Differential Pulse Code Modulation*).
2. Decodificador de vídeo MPEG 4 [69].
3. Digiseal, um sistema para detectar violação em medidores de energia instalados em residências.
4. H.264, um sistema para compressão de vídeos digitais [13].
5. Núcleo do processador de rede R2NP (*Reconfigurable RISC Network Processor*) [14].
6. Sistema de Verificação Automática de Identidade Vocal (SVAIV).
7. Sistema Compressor Sem Perdas para Sinais Biológicos e Imagens médicas (SCSP).
8. Sistema transmissor HDMI (*High Definition Multimedia Interface*).
9. Controlador programável de Motor de Passo (MP).
10. Decodificador de áudio MPEG-2.
11. DCT-2D: codificação de imagens em formato JPEG.
12. Módulo de processamento para a filtragem digital de imagens em tempo real (FDITR).

4.2.3 Projetos Selecionados

De acordo com os dados da Tabela 4.1, quatro projetos de circuitos digitais foram selecionados para a realização de experimentos. Tais projetos são DPCM, o decodificador de vídeo MPEG 4, sistema de verificação automática de identidade vocal e sistema compressor

Tabela 4.1: Classificação dos projetos para realização de experimentos.

Projeto	Dirigido por Coberturas	*Black Box*	Acesso ao Código	Suporte à Medição	**Resultado**
DPCM	Sim	Sim	Sim	Sim	Selecionado
MPEG 4	Sim	Sim	Sim	Sim	Selecionado
Digiseal	Sim	Sim	Sim	Não	Não selecionado
H.264	Sim	Sim	Não	Sim	Não selecionado
RN2P	Sim	Não	Sim	Sim	Não selecionado
SVAIV	Sim	Sim	Sim	Sim	Selecionado
SCSP	Sim	Sim	Sim	Sim	Selecionado
HDMI	Sim	Sim	Não	Sim	Não selecionado
MP	Sim	Sim	Não	Sim	Não selecionado
MPEG 2	Sim	Sim	Não	Sim	Não selecionado
DCT-2D	Sim	Sim	Não	Sim	Não selecionado
FDITR	Sim	Sim	Não	Sim	Não selecionado

sem perdas para sinais biológicos e imagens médicas. Tais sistemas foram desenvolvidos usando-se as metodologias de verificação funcional VeriSC ou BVM (*Brazil-IP Verification Methodology*).

O Digiseal não foi utilizado porque o DUV do mesmo foi codificado em Verilog. O fato de estar em Verilog inviabilizou a aplicação de outras ferramentas para avaliar os benefícios providos pela técnica proposta neste documento. Tampouco foi possível usar o sistema H.264 porque o acesso ao projeto está restrito a um grupo de universidades do qual a Universidade Federal de Campina Grande não faz parte. Já o projeto do Núcleo do Processador de Rede R2NP não foi selecionado por causa da disciplina de verificação empregada, que é majoritariamente *white box*.

Os sistemas HDMI, Motor de Passo, Decodificador de áudio MPEG-2, DCT-2D e filtro digital de imagens em tempo real não foram selecionados porque o acesso aos códigos dos mesmos não foi concedido por seus autores.

4.3 Visão geral dos projetos selecionados

4.3.1 O decodificador de vídeo MPEG 4

O decodificador de vídeo MPEG 4 é um sistema desenvolvido pelo Laboratório de Arquiteturas Dedicadas (LAD) da Universidade Federal de Campina Grande (UFCG). Este sistema foi desenvolvido no surgimento do Brazil-IP, um programa que, entre outras coisas, promove a formação de recursos humanos na área de circuitos integrados. O referido MPEG 4 já foi convertido em *chip* de silício, sendo aprovado na primeira rodada de fundição [69]. Segundo os membros do LAD-UFCG, quando esta fundição ocorreu, o MPEG 4 era o *chip* mais complexo desenvolvido por uma universidade brasileira.

O diagrama de blocos deste sistema pode ser visto na Figura 4.1. O vídeo decodificado é recebido pelo módulo de demultiplexação de vídeo. A partir deste bloco, o vídeo passa por um processo de estimação de movimento e de decodificação de textura. Por último, o vídeo passa por um processo de composição para ser finalmente apresentado. Cada etapa do processamento pode ser realizada seguindo-se mais de um algoritmo. Por esta razão, existe a distinção entre conexão de dados e conexão de configuração na Figura 4.1. A conexão de dados diz respeito ao fluxo de vídeo propriamente e a conexão de configuração diz respeito aos parâmetros de configuração que devem ser usados na decodificação do fluxo de vídeo. No bloco SI, por exemplo, o fluxo de vídeo pode ser processado seguindo-se três algoritmos diferentes. Qual algoritmo que deve ser utilizado é determinado pelos parâmetros de configuração que este bloco recebe.

O Modelo de Referência usado para verificar o MPEG4 foi o Software XVID [2]. Este software, originalmente desenvolvido em linguagem de programação C, não foi reusado como em sua codificação original. Ele foi refatorado para que sua hierarquia de blocos se tornasse adequada à verificação funcional da implementação em RTL deste sistema. O ambiente de verificação e DUV do MPEG 4 foram desenvolvidos em SystemC. O protocolo de comunicação com os blocos não segue um padrão definido pela indústria. Este protocolo foi desenvolvido pelos próprios pesquisadores do LAD.

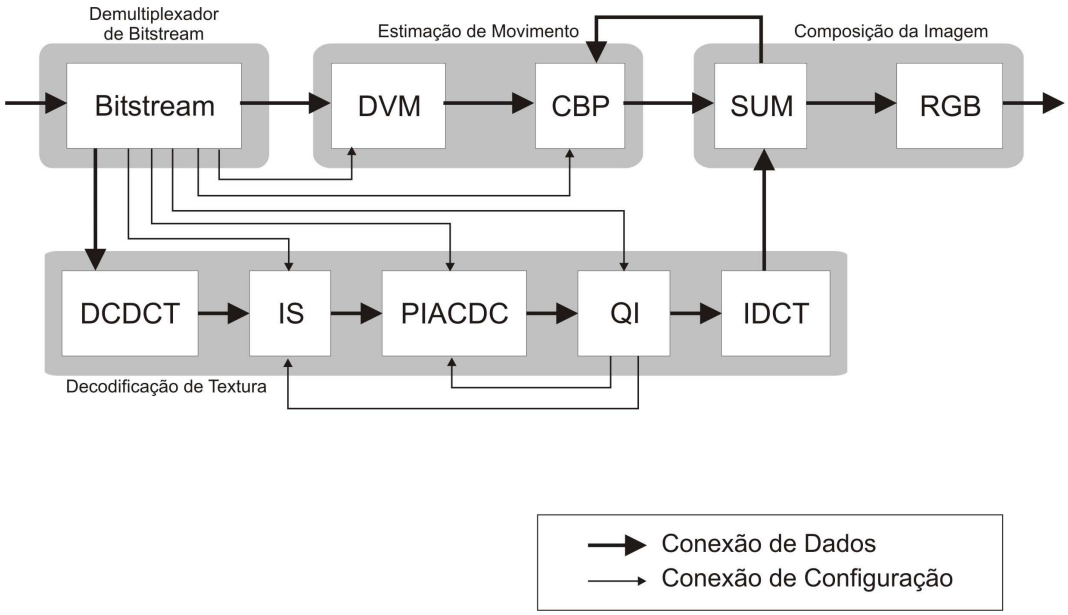

Figura 4.1: Diagrama de blocos do decodificador de vídeo MPEG 4.

4.3.2 O sistema de verificação automática de identidade vocal

O sistema de verificação automática de identidade vocal tem como objetivo principal identificar, a partir da voz, usuários em outros sistemas cujo acesso só pode ser realizado por usuários previamente cadastrados e autorizados. Assim, a partir da sua voz, uma identidade é alegada pelo usuário e o verificador determina se a identidade alegada é verdadeira ou não.

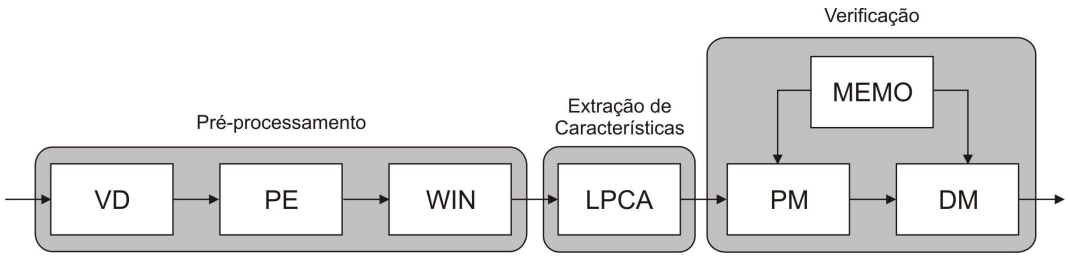

Figura 4.2: Diagrama de blocos do Verificador de Identidade Vocal.

A Figura 4.2 é uma ilustração do diagrama de blocos deste sistema. O processo de verificação é baseado na extração de parâmetros de voz de um locutor para a definição de um modelo que preserva as características vocais que diferenciam o referido locutor de outros indivíduos. A partir desta extração, a verificação consiste em comparar padrões vocais. O locutor será aceito ou rejeitado comparando-se o padrão obtido pela identidade vocal fornecida com um padrão de referência previamente armazenado [39].

Ao longo do Capítulo 3, assumiu-se uma verificação *black-box* no qual os critérios de cobertura são especificados em componentes TDriver e TMonitor. No caso do sistema de identidade vocal, os critérios de cobertura estão todos no modelo de referência. Assim, para viabilizar a utilização deste sistema nos experimentos, foi necessário tirar a especificação de cobertura do modelo de referência, colocando-a em componentes TDriver e TMonitor.

4.3.3 O sistema compressor sem perdas para sinais biológicos e imagens médicas

O monitoramento de pacientes por meio de registro de sinais biológicos ou imagens médicas costuma gerar elevadas quantidades de dados. Além disso, em muitos casos, é fundamental que os equipamentos que realizam tal monitoramento sejam de pequenas dimensões e de baixo consumo. Um exemplo nesse sentido são os *holters*, que são equipamentos portáteis para armazenamento de dados contínuos do ritmo e da freqüência cardíaca por um período médio de 24. Os *holters* são indispensáveis para o monitoramento e diagnóstico preciso de certas doenças cardíacas.

Diante da necessidade de portabilidade e da grande quantidade de dados produzida, os equipamentos para armazenamento de sinais biológicos, de uma forma geral, necessitam de alguma forma de compressão de dados. Além disso, pode ser o caso em que alterações nos dados obtidos na monitoração sejam inaceitáveis, exigindo assim um esquema de compressão sem perdas para a reconstrução perfeita do sinal original a partir do sinal comprimido. Na compressão sem perdas, o sinal é integralmente preservado, procurando-se unicamente reduzir a redundância na representação dos dados. A Figura 4.3 é uma ilustração do diagrama de blocos do sistema compressor de sinais biológicos.

Figura 4.3: Diagramas de blocos do sistema compressor de sinais biológicos e imagens médicas.

4.4 Resultados Experimentais

Os resultados experimentais obtidos com os projetos selecionados serão discutidos nas subseções seguintes. A apresentação dos experimentos segue a ordem das questões declaradas na Seção 4.1.

4.4.1 Questão 1: análise da quantidade de cenários explorados

4.4.2 Ocorrência no MPEG4

Em relação à descoberta de novos critérios de cobertura, o principal resultado obtido foi na integração de blocos de projeto do sistema MPEG 4. Na integração envolvendo os blocos *Bitstream* e PIACDC/QI, foi possível melhorar os critérios de coberturas e fazer com que novos cenários de simulação fossem exercitados. Esta integração está ilustrada nas figuras 4.4 e 4.5. A Figura 4.4 é a ilustração dos ambientes de verificação de cada um dos blocos. No item *a* da Figura 4.5, que foi obtida a partir da Figura 4.1, é possível perceber que PIACDC e QI constituem dois blocos separados do MPEG 4. De fato existem estes dois blocos codificados em RTL. No entanto, não existe um modelo de referência para cada um deles. Não foi possível decompor o código do Xvid para que houvesse um modelo de referência para cada um deles. Assim, na prática, a verificação funcional dos blocos PIACDC e QI ocorre como se os dois tivessem sido integrados em único bloco.

O ambiente de verificação da integração dos blocos *Bitstream* e PIACDC/QI fica como no item *b* da Figura 4.5. Por parte do bloco PIACDC/QI, dois componentes do tipo TDriver são perdidos. Por parte do bloco *Bitstream*, dois componentes do tipo TMonitor são perdidos. Além disso, parte do gerador de estímulos do PIACDC/QI é eliminada, pois o bloco vizinho passa a fazer a geração de tais estímulos.

A análise dos componentes que são descartados na integração revelou que certos cenários de simulação não foram considerados na verificação funcional do bloco PIACDC. O incremento na especificação da verificação do bloco PIACDC foi avaliado por meio de cobertura estrutural. Usando a ferramenta *gcov* para programas C, três baterias de verificação funcional foram realizadas. Em cada uma delas, mediu-se a cobertura de linha de código e cobertura de ramificação. A primeira bateria foi realizada usando-se o ambiente de verificação original

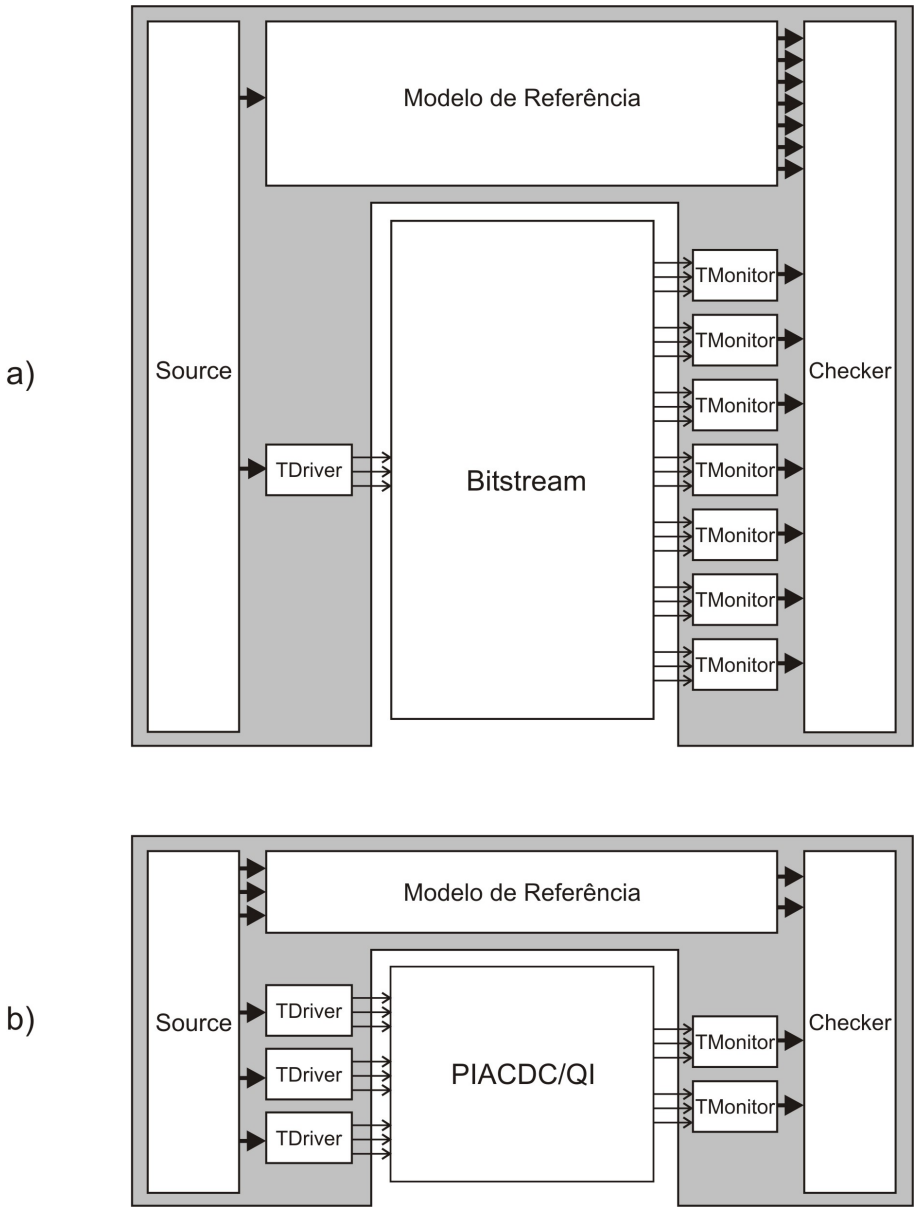

Figura 4.4: Ambientes de verificação de blocos do MPEG 4: a) bloco Bitstream e b) bloco PIACDCQI.

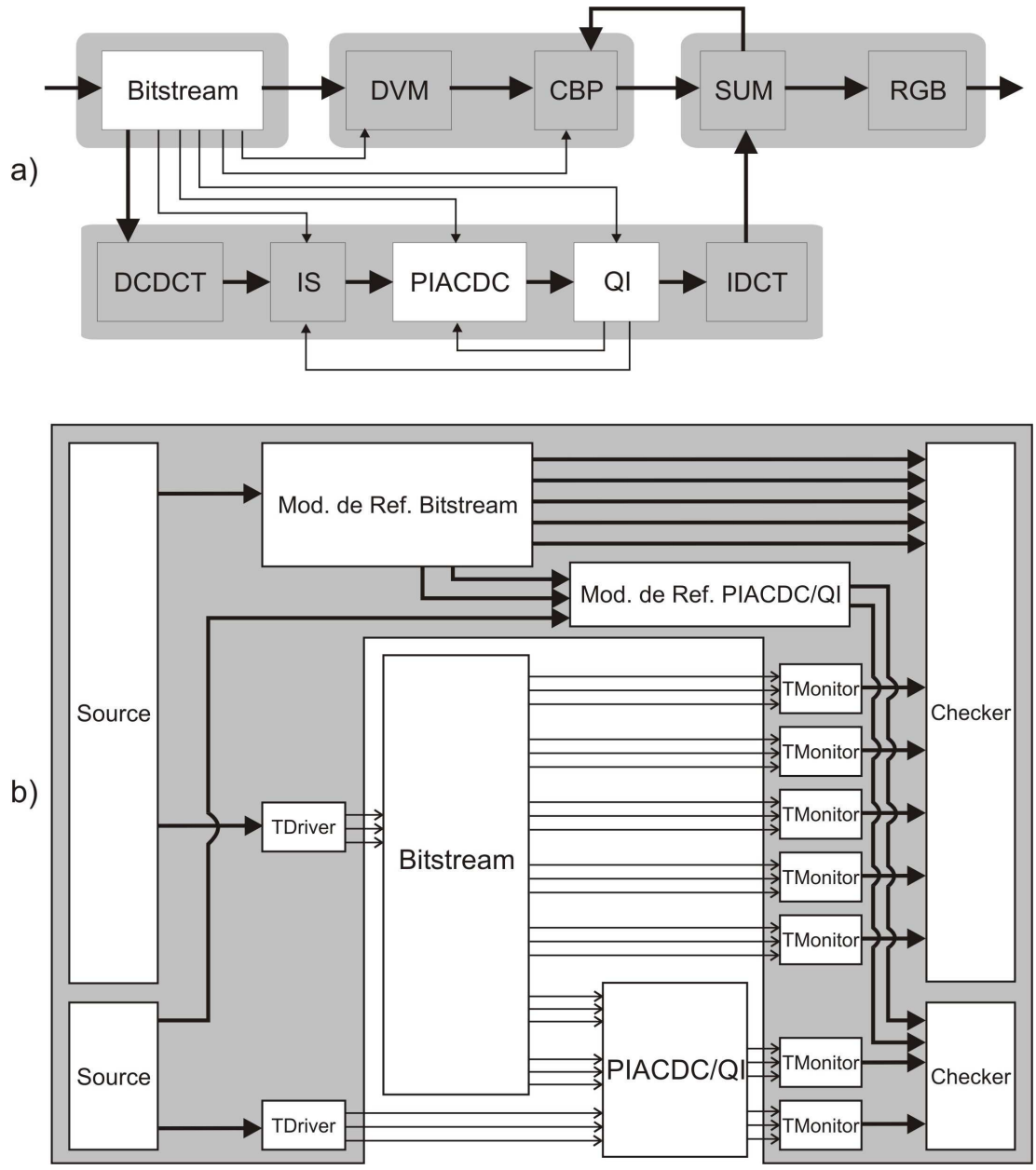

Figura 4.5: Integração dos blocos Bitstream e PIACDCQI: a) diagrama de blocos obtido a partir da Figura 4.1 e b) ambiente de verificação.

do bloco PIACDC/QI. A segunda bateria considerou não apenas o bloco PIACDC/QI, mas a integração deste com o *Bitstream*, tal como ilustrado no item *b* da Figura 4.5. A terceira foi realizada com o bloco PIACDC/QI, mas com o novo ambiente de verificação funcional. Os números da análise de cobertura de linhas de código e de ramificação obtidos em cada uma das baterias estão na Tabela 4.2.

Tabela 4.2: Resultados experimentais usando análise de cobertura estrutural para os blocos QI e PIACDC do MPEG 4.

	Cobertura de Linha de Código			**Cobertura de Ramificação**		
	Spec. Original	Integração	Spec. Nova	Spec. Original	Integração	Spec. Nova
QI	78,93	71,85	82,37	48,91	43,06	50,77
PIACDC	97,83	91,97	99,28	82,87	76,36	84,11

Os piores valores de cobertura estrutural foram obtidos com o ambiente de verificação da integração dos blocos. Estes valores são de certa forma esperados porque diversos componentes que podem conter critérios de cobertura não são reusados. No entanto, quando estes componentes que não são reusados são considerados para melhorar especificação de cada bloco isoladamente, obteve-se uma cobertura maior que a obtida usando-se a especificação original de cada bloco.

A análise de cobertura estrutural somente através do percentual de cobertura obtido pode ocultar uma informação importante para a validação do trabalho proposto. Embora a verificação funcional com o ambiente de verificação da integração dos blocos tenha produzido os piores valores de cobertura estrutural, existe a possibilidade de que a parte coberta na integração seja justamente a parte que não foi coberta na verificação isolada dos blocos. Neste caso, a verificação isolada de cada bloco e a verificação da integração de ambos estariam funcionando de forma complementar, como se cada uma fosse responsável por cobrir uma parte do código.

Ao analisar em detalhes as partes cobertas em cada bateria de verificação, percebe-se que a verificação da integração não cobre as partes do código que ficaram descobertas na verificação individual de cada bloco. As figuras 4.6 e 4.7 representam graficamente as partes cobertas nos blocos QI e PIACDC respectivamente. Existem três barras horizontais em cada figura, uma para cada especificação utilizada na verificação funcional. A primeira barra é

referente à verificação funcional do bloco com sua especificação original. A segunda barra é referente à verificação do bloco com a especificação que permanece nos componentes reutilizados na verificação da integração. A terceira barra diz respeito à verificação com as especificações que foram obtidas pela aplicação da abordagem proposta neste documento. Os números abaixo de cada barra representam o número da linha do bloco de projeto. Os trechos em branco em cada barra representam partes do código que não foram exercitadas. A partir do alinhamento das três barras, é possível perceber que todas as partes cobertas com o ambiente de verificação da integração são também cobertas na verificação individual de cada bloco. Novos trechos de código são exercitados quando se utiliza a nova especificação, obtida pela análise dos componentes que são descartados na construção do ambiente de verificação da integração. Os resultados obtidos na verificação da integração após aplicação da abordagem proposta são os mesmos obtidos na verificação da integração antes da aplicação da abordagem proposta. Isto porque a análise dos componentes descartados revelou novas especificações para os blocos PIACD e QI, enquanto as especificações para o bloco Bitstream permaneceram as mesmas. Conseqüentemente, para este caso, o ambiente de verificação da integração não se altera após aplicação da abordagem proposta.

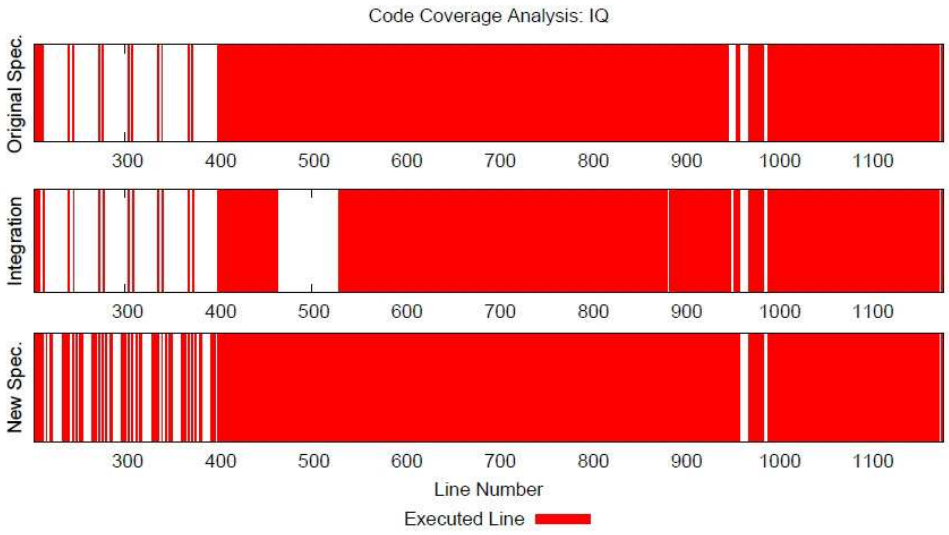

Figura 4.6: Representação gráfica da análise de cobertura de código do bloco QI.

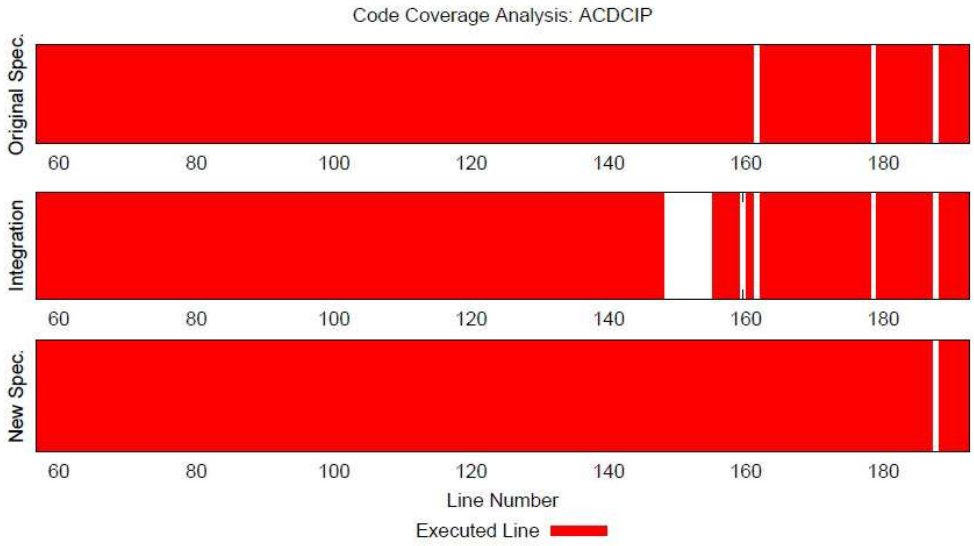

Figura 4.7: Representação gráfica da análise de cobertura de código do bloco PIACDC.

4.4.3 Ocorrência no sistema de identificação vocal

No sistema de identificação vocal (vide Figura 4.2), foi possível sugerir alterações em três partes do projeto. Entre os blocos PE e WIN, foi detectado que a cobertura na saída do bloco PE podia ser alterada para ficar de acordo com a especificação do WIN. Adicionalmente, foi detectado também que a cobertura na saída do bloco WIN poderia ser alterada para ficar de acordo com a especificação do bloco LPCA. Os novos critérios de cobertura dos blocos PE e WIN não exigiram uma reconfiguração dos geradores de estímulos dos referidos blocos. O terceiro caso foi entre os blocos LPCA e PM. A especificação de geração de estímulos do bloco PM revelou que alguns cenários de simulação não estavam sendo considerados na verificação do bloco LPCA. Tais cenários foram especificados para o referido bloco, mas a configuração original do gerador de estímulos do LPCA não foi suficiente para satisfazer os novos critérios. De forma independente do desenvolvimento do sistema pelos engenheiros do LAD, tentou-se configurar o gerador de estímulos para satisfazer os novos critérios de cobertura, mas não foi possível obter sucesso.

Diferentemente do experimento com os blocos do MPEG 4, mesmo antes das alterações nos critérios de cobertura funcional, a cobertura de código dos blocos do sistema de identidade vocal já alcançava 100%. Assim, não foi possível mensurar o melhoramento proposto em função das métricas que foram utilizadas no MPEG 4.

4.5 Questão 2: análise de erros detectados na verificação funcional

Em relação à análise de erros detectados, o principal resultado obtido também foi na integração dos blocos *Bitstream* e PIACDC/QI do MPEG 4, ilustrada nas figuras 4.4 e 4.5. A verificação funcional do bloco PIACDC/QI, após a inserção dos novos critérios de cobertura e configuração de geradores de estímulos, resultou na detecção de erros. O componente *Checker* do referido bloco passou a acusar a divergência entre os resultados produzidos pelo Modelo de Referência e pelo DUV. Esta divergência, no entanto, não garante que o DUV do bloco PIACDC/QI não esteja de acordo com a especificação. Em outros blocos do MPEG 4, por exemplo, a verificação funcional apontou falso negativos porque a implementação do Xvid não estava de acordo com a especificação.

A análise de mutantes em VeriSC foi realizada conforme proposto por Cunha [30]. A Figura 4.8 é uma ilustração de como é organizado o ambiente de verificação para análise de mutantes em VeriSC. A análise de mutantes não depende do DUV. O DUV é substituído por uma réplica do Modelo de Referência, sendo que os mutantes são gerados sobre esta réplica. Ao redor do modelo de referência mutante existe um par de TMonitor e TDriver porque o DUV que foi substituído opera no nível de sinais e modelo de referência que o substitui opera em nível de transações.

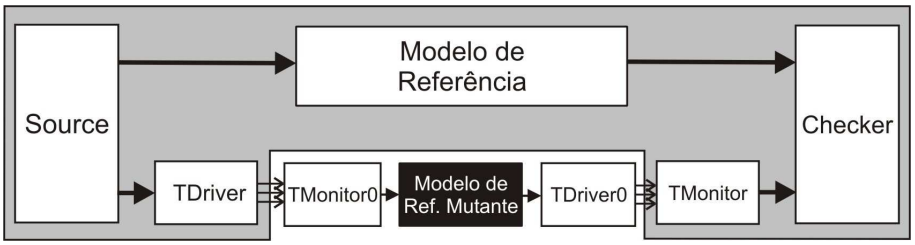

Figura 4.8: Ambiente de verificação para análise de mutantes em VeriSC.

A metodologia VeriSC não possui uma ferramenta própria para a geração de mutantes do Modelo de Referência. A análise de mutantes proposta por Cunha faz uso da ferramenta Proteum [56], desenvolvida originalmente para ser aplicada em programas codificados em linguagem de programação C.

Embora a abordagem de Cunha tenha sido avaliada usando o bloco de projeto IDCT do

decodificador de vídeo MPEG 4 (vide Figura 4.1), não foi possível aplicar todos os modelos de referência deste decodificador à ferramenta Proteum. Após dedicação de esforço considerável à aplicação dos demais blocos, chegou-se a conclusão de que não seria possível realizar análise de mutantes por dois problemas cujas soluções estão fora do escopo deste trabalho: problemas com a ferramenta de geração de mutantes e problema com a implementação do modelo de referência. A versão de Proteum fornecida pelos autores apresentou diversos problemas, sendo que não foi possível gerar mutantes para todo o sistema Xvid, que foi usado como modelo de referência do MPEG 4. É importante observar que foram feitas várias tentativas sem sucesso, inclusive com a ajuda dos próprios autores da ferramenta Proteum. O segundo problema não está relacionado à ferramenta Proteum, mas à implementação do Modelo de Referência. Para aplicação da análise de mutantes, é necessário que o modelo de referência seja reentrante. Em outros termos, isto significa que é necessário que o modelo de referência tenha capacidade de ser executado concorrentemente de forma segura. Porém, nem todos os modelos de referência do MPEG 4 são garantidamente reentrantes.

Para o sistema DPCM, apresentado no Capítulo 3, foi possível empregar análise de mutantes para avaliar o melhoramento obtido pela aplicação da técnica apresentada neste documento. Para tanto, primeiro gerou-se mutantes para o modelo de referência do sistema. Em seguida, duas baterias de verificação funcional foram realizadas. Na primeira bateria foi usada a especificação original de cada bloco sistema. Na segunda bateria, foi usada a especificação que foi melhorada por meio da análise dos componentes do ambiente de verificação que são descartados na integração. Por último, ocorreu a comparação dos resultados obtidos em cada uma das baterias. Estes resultados estão apresentados na Tabela 4.3.

Tabela 4.3: Resultados experimentais usando análise de mutantes para sistema DPCM.

	Especificação Original		**Especificação Nova**	
	Bloco DIFF	Bloco SAT	Bloco DIFF	Bloco SAT
Mutantes	44	100	44	100
Mutantes Mortos	44	67	44	94
Mutantes Vivos	0	33	0	6
Mutantes Equivalentes	0	4	0	4
Mutantes Não Equivalentes e Vivos	0	29	0	2

A linha *Mutantes* da Tabela 4.3 se refere ao número de mutantes gerados para cada um dos blocos. Estes números são iguais nas duas baterias porque a geração de mutantes não depende da especificação dos critérios de cobertura, a geração depende apenas do Modelo de Referência. A linha *Mutantes Mortos* se refere ao número de mutantes que de fato não estão de acordo com a especificação e que foram assim notados pelo componente Checker. Para o bloco DIFF, por exemplo, a verificação funcional das duas baterias foi capaz de *matar* todos os mutantes que não são equivalentes ao sistema original. A linha *mutantes vivos* contém o número de mutantes que a verificação funcional atestou estar em conformidade com o sistema original. A verificação funcional com o ambiente de verificação original do bloco SAT constatou que 33% dos mutantes estão de acordo com a especificação do sistema. Já a verificação funcional com o novo ambiente de verificação constatou que somente 6% dos mutantes estão de acordo com a especificação do sistema. Isto significa que a especificação do novo ambiente de verificação funcional é capaz de detectar mais erros que a especificação original. A linha seguinte contém a informação de que 4% dos mutantes do bloco SAT são equivalentes ao programa original. Por último, com a especificação original 29% dos mutantes do bloco SAT permaneceram vivos, mesmo não sendo equivalentes ao programa original. Mas com a nova especificação, o mesmo acontece somente com 2% dos mutantes do referido do bloco.

4.6 Questão 3: análise do reuso na integração

Para avaliar o impacto do refatoramento do ambiente de verificação, discutido na Seção 3.6, duas integrações envolvendo blocos do MPEG 4 foram usadas como base para realização dos experimentos. Uma integração envolve os blocos DCDCT e SI, e a outra envolve os blocos Bitstream e PIACDC/QI. Para cada integração, foram considerados dois tipos de ambiente de verificação. O primeiro tipo é organizado tal como prescreve VeriSC, isto é, com um único Source e um único Checker, independentemente do número de interfaces de entrada e de saída do bloco. O segundo tipo de ambiente de verificação considera um Source para cada interface de entrada e um Checker para cada interface de saída, tal como proposto na Seção 3.6.

O ambiente de verificação da integração dos blocos DCDCT e SI tal como prescreve

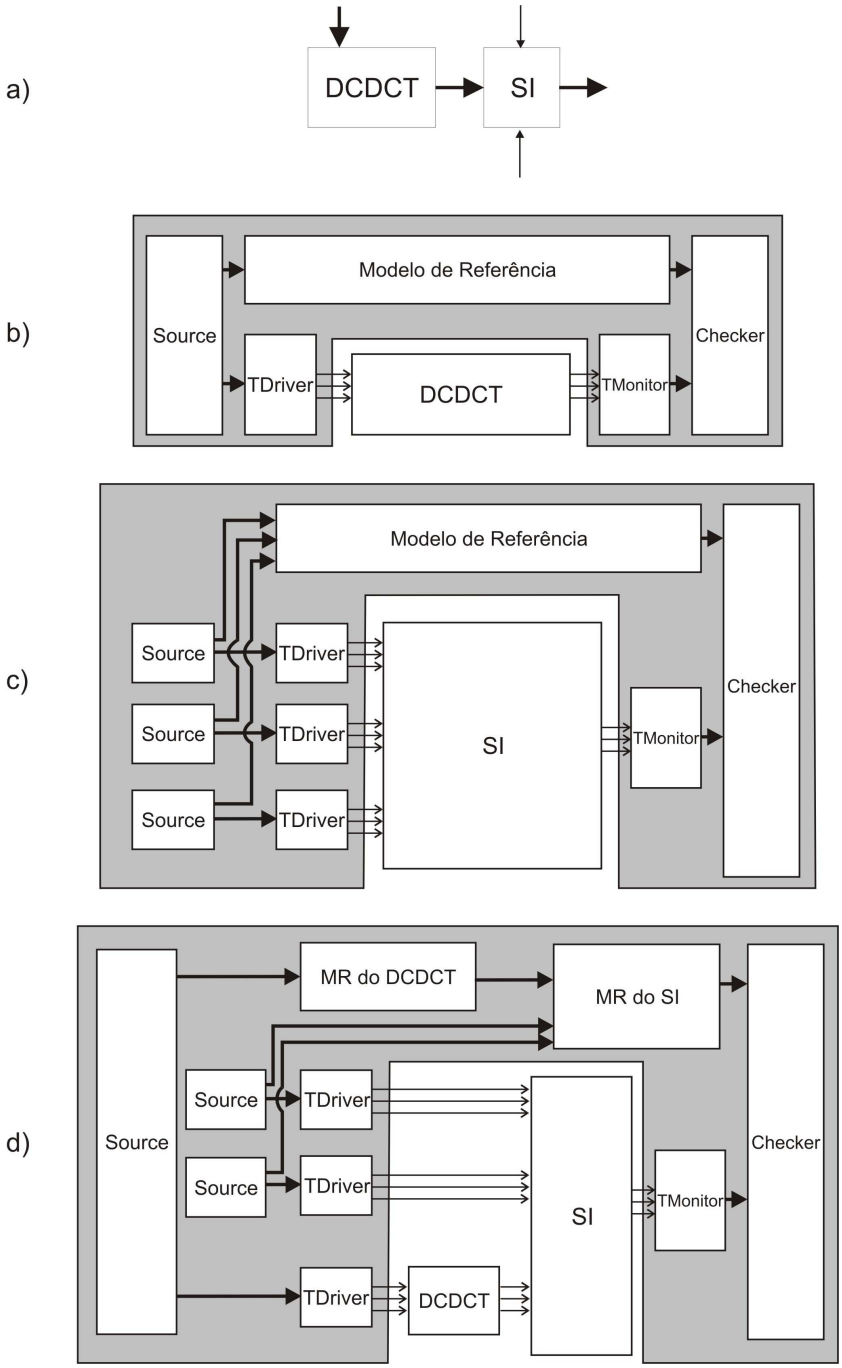

Figura 4.9: Integração dos blocos DCDCT e SI do MPEG 4 com um componente Source para cada interface de entrada: a) Diagrama de blocos da integração; b) Ambiente de verificação do bloco DCDCT; c) Ambiente de verificação do bloco SI e d) Ambiente de verificação da integração dos blocos DCDCT e SI.

VeriSC está ilustrado na Figura 4.12. O custo de se editar os componentes do ambiente de verificação para se obter tal ambiente de verificação está registrado na Tabela 4.4, segunda e terceira colunas. Os arquivos que precisam ser editados são o Source e o TB (*TestBench*), que é responsável por englobar todos os componentes do ambiente de verificação. Três tipos de edição foram considerados: alteração, remoção e inserção de uma linha. A alteração significa que o comando foi mantido, mas com algum tipo de alteração. A remoção denota que comando foi completamente removido. Por último, a inserção significa que um novo comando foi inserido no arquivo.

O ambiente de verificação da integração dos blocos DCDCT e SI que considera um componente Source para cada interface de entrada está ilustrado na Figura 4.9. O custo de se editar os componentes do ambiente de verificação para se obter tal ambiente de verificação também está registrado na Tabela 4.4, mas nas duas últimas colunas. De fato, o único arquivo que precisa ser editado é o TB. O bloco SI tem três interfaces de entrada, portanto, possui três componentes do tipo Source. Mas, a integração requer somente dois destes componentes. Quando estes dois componentes são reusados na integração, eles não precisam ser editados. Assim, comparando-se o custo de edição das duas integrações, percebe-se que o custo de edição quando se utiliza um componente Source para cada interface de entrada é menor que o custo quando se utiliza o ambiente de verificação original de VeriSC.

	Ambiente Original		Novo Ambiente	
	Source	TB	Source	TB
Linhas Alteradas (%)	8,0	6,1	0	4,9
Linhas Removidas (%)	42,0	14,7	0	18,9
Linhas Inseridas (%)	2,0	38,7	0	50,8

Tabela 4.4: Dados sobre as edições em arquivos do ambiente de verificação do SI para se obter a integração dos blocos DCDCT e SI.

A integração dos blocos Bitstream e PIACDC/QI serve para avaliar o benefício de se usar um Source para cada interface de entrada e um Checker para cada interface de saída. O ambiente de verificação da integração de tais blocos tal como prescreve VeriSC está ilustrado na Figura 4.5. Repare que dois TDrivers do PIACDC/QI e dois TMonitors do Bitstream não podem ser considerados na integração. O custo de se editar os componentes do ambiente de

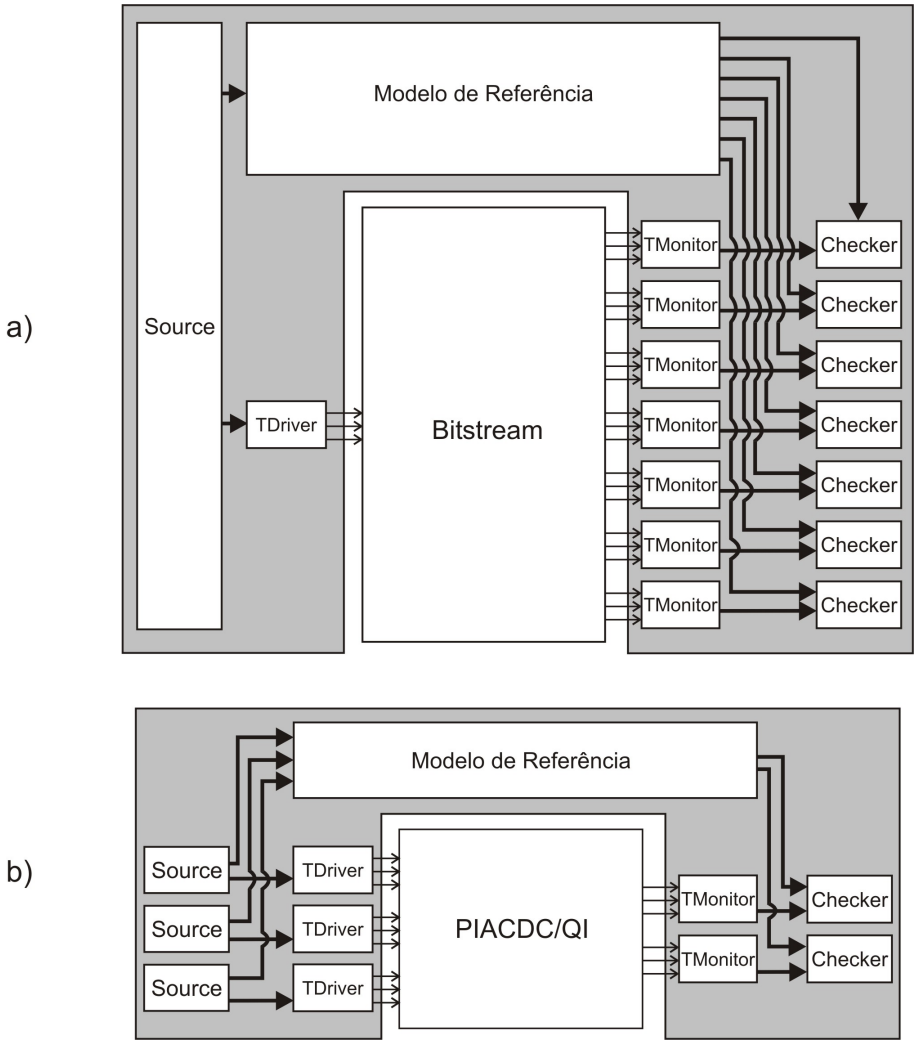

Figura 4.10: Ambientes de verificação de blocos do MPEG 4 com múltiplos geradores de estímulos e múltiplos comparadores: a) bloco Bitstream e b) bloco PIACDCQI.

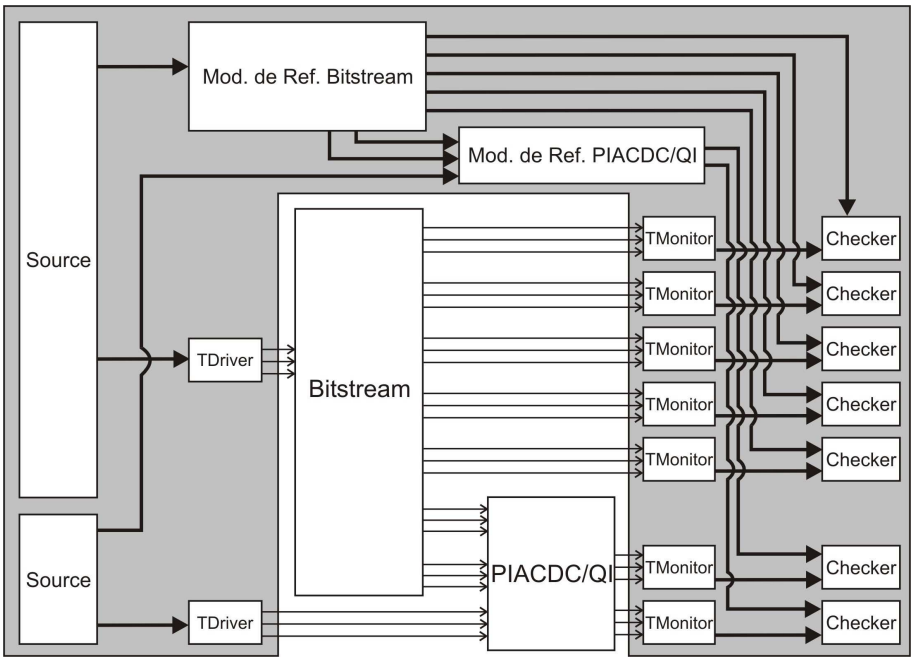

Figura 4.11: Ambientes de verificação da integração dos blocos Bitstream e PIACDC/QI do MPEG 4 com múltiplos geradores de estímulos e múltiplos comparadores.

verificação do bloco Bitstream e do PIACDC/QI para se obter tal ambiente de verificação está registrado na Tabela 4.5, na segunda, terceira e quarta colunas.

O ambiente de verificação da integração dos blocos Bitstream e SI que considera um componente Source para cada interface de entrada e um componente Checker para interface de saída está ilustrado nas figuras 4.10 e 4.11. O custo de se editar os componentes do ambiente de verificação para se obter tal ambiente de verificação também está registrado na Tabela 4.5, nas três últimas colunas. O único arquivo que precisa ser editado é o TB. O bloco Bitstream possui sete interfaces de saída, conseqüentemente, possui setes componentes do tipo Checker. Mas a integração requer cinco destes componentes, pois duas interfaces de saída passam a alimentar o bloco PIACDC/QI. Quando os cinco componentes Checker são reusados na integração, eles não precisam ser editados. Assim, comparando-se o custo de edição das duas integrações, percebe-se que o custo de edição quando se utiliza um componente Checker para cada interface de saída também é menor que o custo quando se utiliza o ambiente de verificação original de VeriSC.

	Ambiente Original			Novo Ambiente		
	Source	Checker	TB	Source	Checker	TB
Linhas Alteradas (%)	1,9	0,0	24,3	0,0	0,0	26,0
Linhas Removidas (%)	59,0	25,6	15,8	0,0	0,0	16,2
Linhas Inseridas (%)	1,0	0,0	127,2	0,0	0,0	125,5

Tabela 4.5: Dados sobre as edições em arquivos do ambiente de verificação para se obter a integração dos blocos DCDCT e SI.

4.7 Questão 4: Análise do tempo gasto na integração.

No sistema MPEG 4, foi detectada uma situação em que a cobertura funcional individual dos blocos é alcançada integralmente, mas durante a integração, os critérios de cobertura remanescentes deixaram de ser alcançados em sua totalidade [71]. A Figura 4.12 é uma ilustração desta integração. Na integração dos DUVs, um componente TMonitor, situado na saída do DCDCT, deixa de existir porque é realizada a ligação direta dos sinais entre DCDCT e SI. Analogamente, o bloco TDriver, situado na entrada do SI deixa de existir. O Source do bloco DCDCT pode ser reusado da mesma maneira que ele foi usado na verificação do bloco isoladamente. O mesmo não é verdade para o Source do bloco SI. Este gerador de estímulos precisa ser restringido, dado que parte do seu comportamento será executada pelo bloco da integração.

Inevitavelmente, alguns componentes do ambiente de verificação são perdidos. Nesta integração, dois deles foram descartados: o TDriver do bloco SI, correspondente à interface de comunicação com o bloco DCDCT, e o componente TMonitor do bloco DCDCT correspondente a sua única interface de saída. Juntos com estes componentes, também são perdidas as especificações de critérios de cobertura contidas nos mesmos. Neste caso de integração, o sistema passa a ter 3 interfaces de entrada. No somatório total, antes da integração existiam 4 interfaces de entrada. Na saída, a integração possui uma única interface de saída. Antes existiam duas, uma para cada bloco.

Na prática, o descarte de componentes do ambiente de verificação e a restrição de geradores de estímulos podem levar a impasses que penalizam o projeto. Estes impasses são decorrentes de critérios de cobertura que são 100% alcançáveis na verificação dos blocos

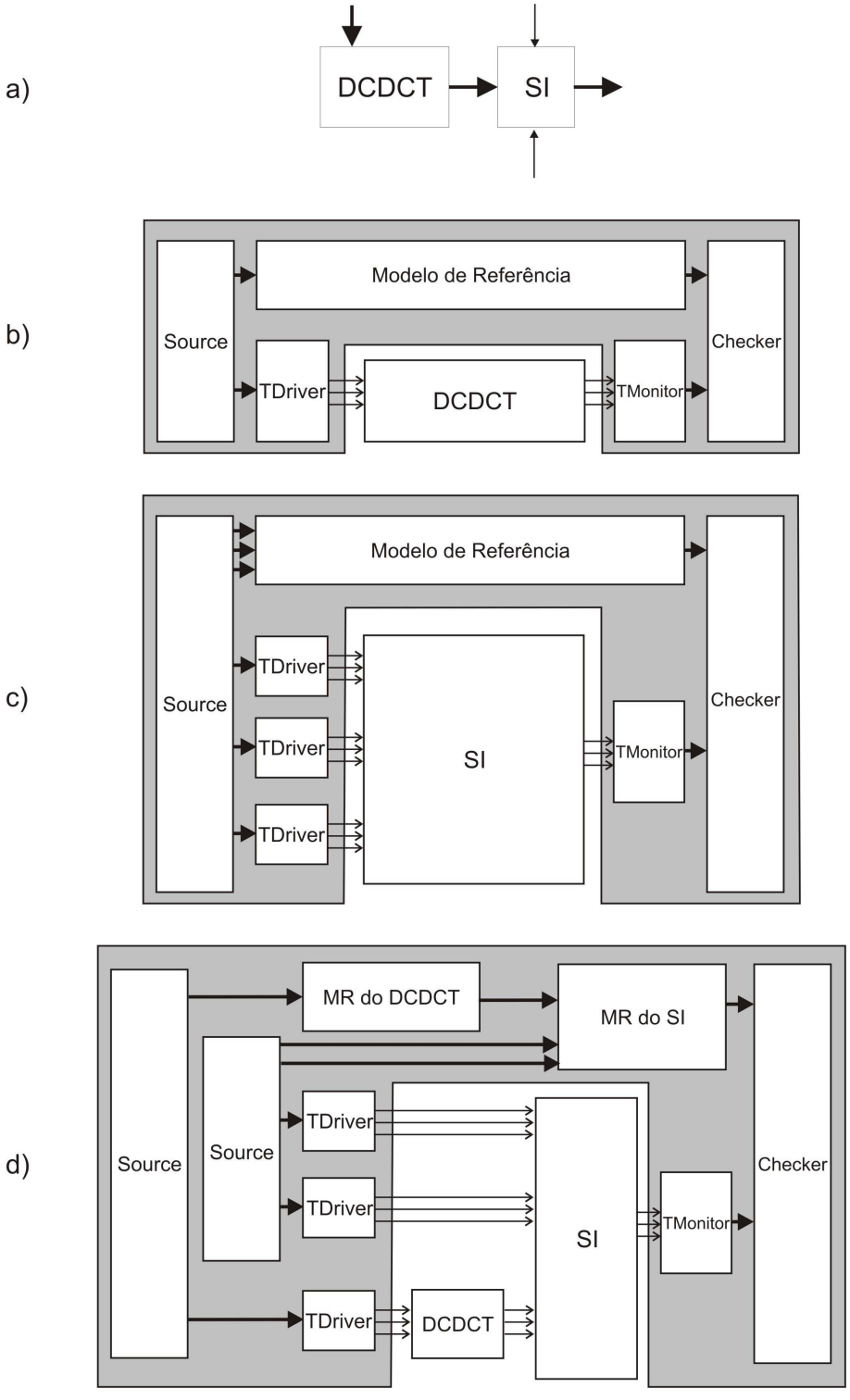

Figura 4.12: Integração dos blocos DCDCT e SI do MPEG 4: a) Diagrama de blocos da integração; b) Ambiente de verificação do bloco DCDCT; c) Ambiente de verificação do bloco SI e d) Ambiente de verificação da integração dos blocos DCDCT e SI.

isoladamente, mas que deixam de ser alcançáveis na integração. Este caso está ilustrado na Figura 4.13. A Figura 4.13 é um gráfico que mostra como a cobertura funcional evolui com o passar do tempo. Os blocos isolados alcançam 100 %, porém a integração de ambos não alcança 100 %.

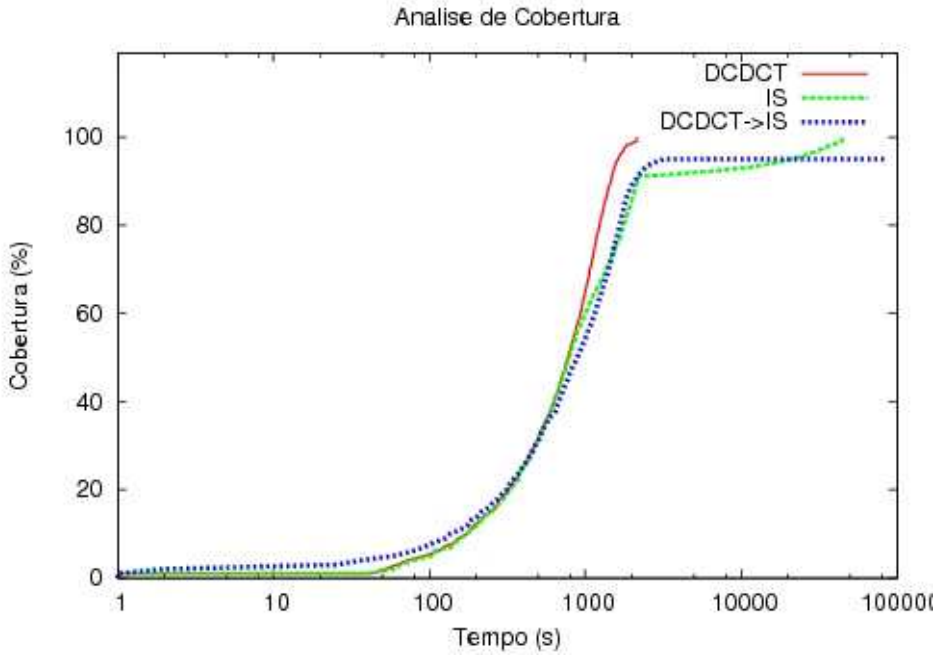

Figura 4.13: Análise de cobertura funcional na integração dos blocos DCDCT e SI do MPEG 4.

Experimentos com os engenheiros LAD

Conforme discutido no Capítulo 1, quando ocorre uma situação em que os critérios de cobertura funcional dos blocos integrados não são integralmente satisfeitos, o engenheiro pode estar diante de uma parte do projeto que não foi devidamente verificada ou então de um projeto cujos critérios de cobertura valem para os blocos individualmente, mas não valem para a integração. Este último caso ocorre, por exemplo, quando um bloco Y foi verificado em relação a um conjunto de funcionalidades C, mas o bloco vizinho que provê entradas para Y só é capaz de prover um conjunto de entradas que cobrem um conjunto de funcionalidades C' sendo que $C' \subset C$.

A realização da integração referente às Figuras 4.12 e 3.6 consumiu 16 horas de dois engenheiros de verificação. As atividades realizadas para se obter tal integração foram: criação

do ambiente de verificação da integração; testes e depuração deste novo ambiente; verificação funcional considerando os geradores de estímulos e critérios de cobertura funcional que foram reusados dos ambientes de verificação originais de cada bloco da integração; constatação de que a verificação funcional não alcançava 100% porque o gerador de estímulos não estava devidamente configurado para que os critérios de cobertura fossem satisfeitos e, por último, reconfiguração dos geradores de estímulos e da cobertura funcional do bloco DCDCT para que os critérios de cobertura fossem satisfeitos.

Para se obter uma noção do custo de diagnosticar que os critérios de cobertura de uma integração não podem ser satisfeitos, foram realizados experimentos envolvendo engenheiros de verificação do LAD-UFCG e Laboratório para a Integração de Circuitos e Sistemas (LINCS) de Campina Grande. Dez engenheiros de verificação foram convidados a participar dos experimentos. Destes dez, somente quatro aceitaram o convite. A integração ilustrada na Figura 4.12 foi utilizada como base para tais experimentos. Reduziram-se apenas alguns critérios de cobertura para que os experimentos não consumissem um tempo maior que o disponibilizado pelos voluntários. Embora nenhum dos quatro voluntários tenha participado de fato do projeto MPEG 4, todos eles já tinham alguma experiência com o referido projeto durante as atividades para a formação dos mesmos.

Nestes experimentos foi observado o tempo que cada engenheiro levou para diagnosticar se os critérios de cobertura da referida integração poderiam ser satisfeitos ou não. Assim, cada engenheiro não sabia *a priori* se a verificação funcional alcançaria ou não 100%. Fazia parte dos experimentos que cada engenheiro averiguasse se os critérios de cobertura seriam satisfeitos ou não. No caso de resposta negativa, ele também deveria apresentar uma justificativa para o fato.

O ambiente de verificação da integração foi passado pronto para os engenheiros e com a garantia de que a integração dos blocos não apresentava nenhum tipo de problema em relação a protocolos de comunicação e interface. A partir de explanação oral, foi esclarecido que projeto sendo verificado era a integração de dois blocos previamente verificados e que ambos tiveram seus critérios de cobertura integralmente satisfeitos. Além disso, foi explicado que o ambiente de verificação da integração foi constituído a partir de componentes usados na verificação de cada bloco. Foi enfatizado que cada engenheiro tinha acesso e permissão para acessar todo o código dos blocos envolvidos para realização dos experimentos.

Os quatro voluntários foram divididos em dois subgrupos com dois componentes cada. Para ambos os subgrupos, os mesmos ambientes de verificação e projetos foram utilizados. A diferença foi que um subgrupo trabalhou de forma *ad hoc* e outro trabalhou de forma sistemática, conforme discutido no Capítulo 3, mas sem apoio ferramental. Assim, os experimentos são uma tentativa de caracterizar não apenas a importância de se proceder de forma metódica à integração, mas também de se construir suporte ferramental para tanto.

Apesar de ter sido enfatizado que era possível acessar o código de todo o projeto, o subgrupo que trabalhou de forma *ad hoc* se apoiou no ambiente de verificação da integração e na informação gráfica provida pela ferramenta de verificação funcional de VeriSC. Em VeriSC, é possível acompanhar a evolução da análise de cobertura funcional através de barras de progressão. Assim, os componentes que não foram reusados na integração não foram considerados pelos engenheiros. O primeiro engenheiro gastou uma hora e dezesseis minutos e argumentou que a verificação não alcançaria 100% porque o componente gerador de estímulos do bloco DCDCT não estava configurado devidamente. Apesar de ter procurado no código uma justificativa mais precisa, ele argumentou que era este o problema porque se a verificação funcional de cada bloco satisfaz integralmente os critérios de cobertura, é esperado que o mesmo ocorra com a integração. O segundo engenheiro também argumentou que verificação não alcançaria 100% de cobertura. A justificativa, no entanto, foi diferente. Como os critérios de cobertura inalcançáveis pertenciam originalmente ao bloco SI, o engenheiro argumentou que os mesmos não deveriam existir, pois eles não eram alcançáveis no sistema resultante da integração dos dois blocos. O tempo gasto por este engenheiro foi de uma hora e três minutos.

Para o subgrupo que procedeu a experimentação usando a técnica apresentada no Capítulo 3, as justificativas fornecidas pelos engenheiros foram precisas. Os dois voluntários deste subgrupo conseguiram determinar exatamente o motivo pelo qual a verificação não alcançava 100%. Eles conseguiram determinar a especificação de cobertura que deveria existir no bloco DCDCT a partir do gerador de estímulos do bloco IS. Em seguida, eles conseguiram determinar uma relação entre a especificação ausente e o critério de cobertura inalcançável da integração. Um engenheiro gastou trinta e um minutos e outro vinte e seis minutos. Assim, por meio da técnica proposta no Capítulo 3, foi possível obter resultados mais precisos e em um menor intervalo de tempo. Apesar de este subgrupo ter consumido quase a metade

do tempo do primeiro, foi possível constatar que as atividades que devem ser realizadas não são triviais e que ferramentas para automatizar a execução das mesmas são necessárias. Um voluntário, por exemplo, precisou usar lápis e papel para fazer anotações e cálculos para diagnosticar o problema.

4.8 Ocorrência de Falsos negativos

Na metodologia VeriSC, o comportamento esperado do projeto é dado pelo Modelo de Referência. Conseqüentemente, quando ocorre uma discrepância entre uma saída produzida pelo Modelo de Referência e uma saída produzida pelo DUV, assume-se que o DUV não está de acordo com a especificação. O processo de depuração se inicia, portanto, com o engenheiro investigando o código do DUV. Embora seja pouco comum, o processo de depuração pode revelar que o DUV de fato está de acordo com a especificação do sistema e que o problema está no Modelo de Referência. Neste caso, a verificação funcional está produzindo um falso negativo.

A técnica proposta no Capítulo 3 é baseada na comparação de conjuntos constituídos a partir de elementos que são descartados na integração. Respeitando-se as especificações de valores ilegais ou que devem ser ignorados na simulação de cada um dos blocos, a comparação visa identificar se um dos blocos está mais bem especificado que o outro. Sendo que *mais bem especificado* significa considerar um conjunto maior de cenários de simulação. Uma vez identificado o bloco que está mais bem especificado, operações são realizadas para que as especificações do outro bloco sejam melhoradas. Assim, é importante observar que as especificações de valores ilegais ou que devem ser ignorados, juntamente com as especificações do bloco que abrangem mais cenários, são consideradas como especificações de *referência*. Tais especificações servem de base para melhorar a especificação do bloco vizinho. Ao assumir tais especificações como referência, torna-se possível a produção de falsos negativos, por exemplo, quando se considera, na geração de estímulos, um conjunto de valores que não pode ser produzido pelo bloco vizinho. Caso assim ocorre no sistema compressor sem perdas para sinais biológicos, descrito a seguir.

4.8.1 Sistema compressor sem perdas para sinais biológicos

A interface de saída do bloco PPM é usada para a comunicação de três dados com o bloco Arith: *low*, *high* e *total*. Em relação ao dado *total*, os valores cobertos estão entre 2 e 508 conforme especificado no Código 12. No entanto, as restrições usadas na geração de estímulos do bloco Arith, apresentada no Código 13, permite que o bloco Arith seja simulado com a dado *total* variando de 1 a 508. De acordo com o apresentado no Capítulo 3, a cobertura do bloco PPM deveria ser melhorada para considerar este novo intervalo.

Ao relatar um novo intervalo de cobertura para o bloco PPM, os engenheiros responsáveis pela verificação do referido bloco argumentaram que o limite inferior do novo intervalo é um valor inalcançável pelo bloco PPM. Em função de atribuições de valores iniciais, não é possível alcançar tal valor. Quando foi reportada a justificativa para se propor o novo intervalo de cobertura, os engenheiros argumentaram que o problema de fato estava na geração de estímulos do bloco Arith. Assim, foi constado que as restrições para a geração de estímulos de Arith é que deveriam ser ajustados para ficar de acordo com os critérios de cobertura do bloco PPM e com a sua própria especificação.

Algoritmo 12 : Cobertura medida na saída do bloco PPM.

```
coverpoint tr_PPM_Saida.total {
     bins tr[] = { [2:508] };
     option.at_least = 30;
 }
```

4.9 Considerações do Capítulo

Este capítulo foi dedicado à apresentação de resultados experimentais obtidos pela aplicação da técnica que visa sistematizar a atividade de integração de blocos de projetos de circuitos digitais. Várias dificuldades foram encontradas no decorrer desta parte do trabalho. Estas dificuldades variaram desde a seleção de projetos para estudos a participação de voluntários em experimentos.

Na parte de experimentos envolvendo engenheiros de verificação, a grande dificuldade foi conseguir voluntários. No caso da equipe da UFCG, os participantes do Brazil-IP são também alunos de graduação. O envolvimento com a graduação e Brazil-IP faz com que o

Algoritmo 13 : Geração de estímulos para o bloco Arith.

```
rand int low;
constraint low_range {
   (low >= 0) && (low < 255);
}

rand int high;
constraint high_range {
   solve low before high;
   (high > low) && (low < 509);
}

rand int total;
constraint total_range {
   solve high before total;
   (total >= high) && (total < 509);
}
```

tempo livre que engenheiro seja bastante reduzido. Embora os experimentos tenham sido planejados para tomar o menor tempo possível e também tenha sido oferecida bonificação para os participantes, o número de voluntários foi consideravelmente pequeno.

Na parte de experimentos envolvendo ferramentas, esforços consideráveis foram dedicados para a utilização de análise de mutantes nos experimentos envolvendo o projeto do MPEG 4. A análise de mutantes foi priorizada em função do seu valor semântico, pois um mutante vivo que não é equivalente ao programa original, significa um erro que o engenheiro de projeto pode cometer e que a verificação funcional não está apta a detectar. É importante considerar que é possível aplicar análise de mutantes no MPEG 4 desenvolvido no LAD. Porém, as atividades necessárias para viabilizar esta aplicação são consideravelmente complexas e fogem do escopo deste trabalho. Por tais motivos, a análise de cobertura estrutural foi usada no lugar da análise de mutantes.

Capítulo 5

Conclusões

Apresentou-se neste trabalho uma abordagem para proceder a verificação funcional na integração de blocos de projetos de circuitos digitais. A verificação funcional na fase de integração é importante porque grande parte dos comportamentos não esperados dos sistemas ocorre em função de interações complexas entre os diversos blocos que os compõem. A dificuldade em realizar a verificação na fase de integração deve-se ao fato de que à medida que os blocos são integrados, o número de combinações de possíveis interações entre eles cresce exponencialmente. Além disso, o desempenho na execução das simulações decai consideravelmente com o aumento da complexidade dos blocos de projeto.

O trabalho proposto foi desenvolvido tendo VeriSC como metodologia de verificação subjacente. Embora não seja uma metodologia empregada pela indústria, VeriSC possui as características básicas que as metodologias adotas pela indústria possuem. O objetivo do trabalho foi desenvolver uma abordagem de integração de blocos de projeto capaz de permitir o reuso dos componentes do ambiente de verificação e também a detecção erros que podem surgir em função da interação entre os blocos integrados.

5.1 Avaliação dos resultados obtidos

Os resultados obtidos na parte experimental do trabalho sugerem que a técnica proposta pode ser útil no caso de projetos com uma hierarquia de blocos complexa, em que um único bloco pode ser comunicar com mais de um bloco vizinho. O fato de um bloco possuir mais de um bloco vizinho torna a verificação funcional uma atividade complexa, pois o engenheiro pode

não saber como configurar a cobertura funcional para que os cenários de simulação mais importantes sejam exercitados. Aplicações da técnica proposta em sistemas mais simples, como alguns desenvolvidos no Brazil-IP usando-se a metodologia BVM, também revelaram melhoramento na especificação de cobertura dos blocos, porém não foi possível quantificar esta melhoria por meio de outras métricas, tais como cobertura de código e análise de mutantes.

É esperado que o melhoramento obtido durante a integração seja menor para sistemas com diagramas de blocos mais simples, em que cada bloco majoritariamente possui uma única interface de entrada e uma única interface de saída. Isto porque o projeto tem uma complexidade que não compromete a habilidade do engenheiro de especificar critérios de cobertura e geradores de estímulos. Quando for o caso do sistema possuir um diagrama de blocos complexo, em que os blocos possuem mais de uma interface de entrada ou mais de uma de saída, e diversos dados são transmitidos por estas interfaces, é esperado que o melhoramento obtido na integração seja maior. Isto porque a especificação de critérios de cobertura de um bloco pode ser complementada a partir da especificação dos blocos da sua vizinhança.

O falta de conformidade entre o modelo de referência e DUV no projeto do MPEG 4 é um exemplo de que o trabalho proposto pode contribuir com a qualidade dos projetos de circuitos digitais. No entanto, é importante destacar que não é possível afirmar categoricamente que o projeto do MPEG 4 possui um erro. Em outras ocasiões do desenvolvimento deste sistema, falso-negativos foram produzidos pela verificação funcional. Embora não exista documento relatando tais falso-negativos, os engenheiros de tal sistema detectaram que o sistema XVID, usado como modelo de referência, não estava de acordo com a especificação do sistema. Mesmo que o suposto erro encontrado no MPEG 4 seja um caso de falso-negativo, o uso de outras métricas para avaliar a qualidade da verificação funcional permitiu constatar que a análise dos componentes de verificação funcional que são descartados na integração pode exercitar cenários que não foram exercitados na verificação isolada dos blocos.

Experimentos em que engenheiros de verificação conduziram a verificação de blocos integrados também foram realizados. Os resultados obtidos em tais experimentos constituem indícios de que a verificação funcional na integração exige de fato uma abordagem sistemática, pois, no caso de ser realizada de forma *ad hoc*, esta etapa pode consumir desne-

cessariamente os recursos do projeto.

Em relação ao reuso dos componentes do ambiente de verificação, foi detectado que o ambiente de verificação de VeriSC poderia ser refatorado para que certos componentes sejam usados na integração tal como são usados na verificação isolada dos blocos e vice-versa, sem a necessidade de edição dos arquivos de tais componentes.

5.2 Sugestões para Trabalhos Futuros

O trabalho proposto lida somente com uma modalidade de geração aleatória de estímulos. Nesta modalidade, os valores gerados devem respeitar um intervalo de valores previamente definida pelo engenheiro de verificação. Falta ainda suporte ferramental para geração de estímulos a partir de expressões de restrição. Para realização desta parte do trabalho, a biblioteca SCV pode servir de ponto de partida. Isto simplificaria parte considerável do problema porque seria utilizada a mesma ferramenta que é utilizada para geração de estímulos com o uso de expressões de restrição. Assim, a biblioteca SCV seria utilizada para a construção dos conjuntos mencionados no Capítulo 3.

VeriSC tem suporte à geração parcial de ambientes de verificação através da ferramenta eTBc. Esta ferramenta, no entanto, considera que cada ambiente deve possuir um único componente Source e um único componente Checker. Conforme discutido no Capítulo 3, esta característica pode comprometer a interoperabilidade dos componentes do ambiente de verificação. Assim, é importante que eTBc seja capaz de gerar ambientes de verificação com múltiplos sources e múltiplos checkers, de acordo com a quantidade de interfaces de entrada e saída que cada bloco de projeto possui.

A análise dos componentes de verificação que são descartados pode levar ao melhoramento dos critérios de cobertura dos blocos individuais. No entanto, este trabalho não aborda como configurar geradores de estímulos para que os novos critérios de cobertura sejam satisfeitos. Existe uma área específica, chamada de *Coverage Directed Test Generation* (CDG) [37; 76], que trata da configuração automática de geradores de estímulos para que critérios de cobertura sejam satisfeitos. Portanto, outro possível trabalho futuro é a combinação do trabalho proposto com a técnica de CDG para diminuir a intervenção humana na fase de integração dos blocos.

A técnica proposta neste trabalho lida com critérios de cobertura funcional especificados por meio de declarações do tipo *BVE_Cover_Bucket*. Este tipo de declaração permite que valores sobre dados individuais sejam analisados durante a simulação. Outra forma de se medir cobertura funcional é a partir de cobertura cruzada de valores. Na cobertura cruzada, o engenheiro especifica as *combinações* de valores que devem ocorrer durante a simulação. Ela serve para especificar, por exemplo, que "todos os*buffers* devem ficar cheios ao mesmo tempo". Dentre os projetos considerados na parte experimental deste trabalho, nenhum deles continha especificação de cobertura cruzada. Possivelmente em função da dificuldade de se trabalhar com este tipo de cobertura.

Especificações de cobertura cruzada também podem ser melhoradas na fase de integração. O relatório técnico apresentado no Apêndice B é um trabalho preliminar que visa melhorar a especificação de cobertura cruzada na fase de integração. A forma como a integração é analisada não exige a presença de especificações de cobertura cruzada. É necessário apenas que a metodologia tenha suporte ao registro de transações em banco de dados. VeriSC, por exemplo, através da ferramenta eTBc, gera automaticamente o código para registrar os estímulos que passam por cada componente TDriver e TMonitor. Uma vez que os estímulos ficam registrados em uma base de dados, as informações sobre as *combinações* de valores são obtidas por meio de mineração de dados, sem a necessidade de especificar as matrizes de cobertura cruzada.

A técnica utilizada para mineração de dados é a mineração de regras de associação [80]. Uma regra de associação é um padrão $X \rightarrow Y$ tal que X e Y são conjuntos de valores. Uma aplicação típica deste tipo de mineração é a venda cruzada em livrarias virtuais. Na venda cruzada, o sistema automaticamente sugere que o consumidor que comprou o livro X também comprou o livro Y. Assim, a idéia básica de melhorar a cobertura cruzada na fase de integração consiste em minerar regras de associação sobre resultados de saída de um bloco, minerar regras de associação sobre os estímulos de entrada do bloco seguinte e analisar as regras obtidas para julgar se os dois blocos foram devidamente verificados.

Para se obter as regras de associação, dois parâmetros devem ser fornecidos: Suporte e Confidência. O Suporte de uma regra de associação é o percentual de transações em um banco de dados que contém todos os itens listados na regra de associação. A Confidência mede o grau de certeza de uma associação. Em termos estatísticos, trata-se da probabilidade

condicional $P(Y|X)$, isto é, a porcentagem de transações contendo os itens de X que também contém os itens de Y. O trabalho apresentado no Apêndice B precisa evoluir na escolha adequada destes dois parâmetros. Além disso, é preciso a elaborar uma forma de traduzir as informações na forma de regras de associação para critérios de cobertura dos blocos de projeto.

Do ponto de vista de aplicação da técnica proposta neste trabalho, o melhoramento da verificação funcional precisa ainda ser avaliado em situações de reuso de blocos de projeto. Por motivos financeiros, blocos de projetos são desenvolvidos para serem empregados em diversos sistemas. Conseqüentemente, um bloco pode ter uma vizinhança diferente a cada vez que ele é reusado em um sistema. Esta variação de contexto dá margem a um melhoramento da especificação a cada vez que o bloco é empregado em um sistema diferente. Por ter sido desenvolvido no âmbito estritamente acadêmico e com pouca disponibilidade de projetos, não foi possível fazer uma avaliação desta natureza.

Na parte experimental deste trabalho, o ambiente de verificação funcional acusou um erro na verificação funcional do bloco PIACDC/QI, que compõe o decodificador de vídeo MPEG 4. Isto não significa que o projeto do referido bloco de fato possui um erro. Engenheiros que participaram do projeto reportaram que, em alguns blocos do MPEG 4, a verificação funcional apontou falso negativos porque a implementação do Modelo de Referência não estava de acordo com a especificação. Desta forma, é preciso uma investigação mais detalhada para determinar se o bloco de projeto do PIACDC/QI não respeita suas especificações. A dificuldade em fazer tal investigação se deve ao fato de Modelo de Referência e DUV não seguem a mesma granularidade de decomposição. No modelo de referência, PIACDC e QI são uma única implementação, sem decomposição. No DUV, PIACDC e QI são blocos que existem separadamente. No trabalho de Rodrigues *et al* [70], modelos separados para o PIACDC e QI foram produzidos em redes de Petri coloridas [51]. Além disso, conforme relatado no Apêndice A, estes modelos foram desenvolvidos para serem utilizados em um ambiente de verificação tal como prescreve VeriSC. Portanto, os referidos modelos podem servir de base para uma investigação detalhada sobre o erro que foi apontado na parte experimental deste trabalho.

Bibliografia

[1] IEEE standard verilog hardware description language. Technical report, IEEE Computer Society, 2001.

[2] Xvid team. xvid api 2.1 reference (for 0.9.x series)., 2003. http://www.xvid.org/.

[3] Cadence incisive functional verification platform, 2007. http://www.cadence.com/products/functional_ver/index.aspx.

[4] Cpn tools, 2007. http://wiki.daimi.au.dk/cpntools/.

[5] Mentor formal pro, 2007. http://www.mentor.com/products/fv/ev/formalpro/index.cfm.

[6] Synopsys discovery plataform, 2007. http://www.synopsys.com/products/solutions/discovery_platform.html.

[7] Systemc community, 2007. http://www.systemc.org/.

[8] The r-project for statistical computing, 2008. http://www.r-project.org/.

[9] Brazil-ip, 2009. http://www.brazilip.org.br/.

[10] Ovm world: Open verification methodology, 2009. http://www.ovmworld.org/.

[11] Systemverilog hardware description and verification language (hdvl) standard, 2009. http://www.systemverilog.org/.

[12] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. Focs: Automatic generation of simulation checkers from formal specifications. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 538–542, London, UK, 2000. Springer-Verlag.

[13] L.V. Agostini, A.P. Azevedo Filho, V.S. Rosa, E.A. Berriel, T.G.S. Santos, S. Bampi, and A.A. Susin. Fpga design of a h.264/avc main profile decoder for hdtv. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug. 2006.

[14] A. M. AMARAL, M. O. S. SOUZA, and C. A. P. S. MARTINS. Ippc: Intellectual property processor component applied in embedded computer systems. In *VII Microelectronics Students Forum, 2007, Rio de Janeiro. SFORUM: Microelectronics Student Forum - Chip in Rio*, Porto Alegre, Brazil, 2007.

[15] Guido Araújo, Edna Barros, Elmar Melcher, Rodolfo Azevedo, Karina R. G. da Silva, Bruno Prado, and Manoel E. de Lima. A systemc-only design methodology and the cine-ip multimedia platform. *Design Automation for Embedded Systems*, 10(2-3):181–202, 2006.

[16] Ansuman Banerjee, Bhaskar Pal, Sayantan Das, Abhijeet Kumar, and Pallab Dasgupta. Test generation games from formal specifications. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 827–832, New York, NY, USA, 2006. ACM.

[17] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, 1994.

[18] J. Bergeron. *Writing Testbenches Using System Verilog*. Springer, Massachusetts, 2006.

[19] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andy Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[20] Jayanta Bhadra, Magdy S. Abadir, Li-C. Wang, and Sandip Ray. A survey of hybrid techniques for functional verification. *IEEE Des. Test*, 24(2):112–122, 2007.

[21] K. Bilinski, J.M. Saul, and E.L. Dagless. Efficient functional verification algorithm for petri-net-based parallel controller designs. *IEEE Proceedings - Computers and Digital Techniques*, 142(4):255–262, 1995.

[22] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[23] Holger Blume, Thorsten von Sydow, and Tobias G. Noll. Performance analysis of soc communication by application of deterministic and stochastic petri nets. In *SAMOS*, pages 484–493, 2004.

[24] D. S. Brahme, S. Cox, J. Gallo, W. Grundmann, nad W. Paulsen C. N. Ip, J. L. Pierce, J. Rose, D. Shea, and K. Whiting. The transaction-based verification methodology. Technical report CDNL-TR-2000-0825, Cadence Berkeley Labs, August 2000.

[25] M. Braun, W. Rosenstiel, and K.-D. Schubert. Comparison of bayesian networks and data mining for coverage directed verification category simulation-based verification. In *HLDVT '03: Proceedings of the Eighth IEEE International Workshop on High-Level Design Validation and Test Workshop*, page 91, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Inc. Cadence Design Systems. Incisive plan-to-closure methodology: design team verification. technical paper. Technical report, 2005.

[27] E. Clarke, O. Grumber, and D. Peled. *Model Checking*. MIT Press, 1999.

[28] Charles J. Colbourn and Jeffrey H. Dinitz. *Handbook of Combinatorial Designs, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2006.

[29] Shady Copty, Itai Jaeger, Yoav Katz, and Michael Vinov. Intelligent interleaving of scenarios: a novel approach to system level test generation. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 891–895, New York, NY, USA, 2007. ACM.

[30] H. N. Cunha. Análise de mutação aplicada à verificação funcional de ip core. *Dissertação de mestrado - COPIN, UFCG*, 2008.

[31] K. R. G. da Silva. *Uma metodologia de Verificação Funcional para Circuitos Digitais*. PhD thesis, 2007.

[32] K. R. G. da Silva, E. U. K. Melcher, I. Maia, and H. do N. Cunha. A methodology aimed at better integration of functional verification and RTL design. *Design Automation for Embedded Systems*, 10(4):285–298, 2007.

[33] Karina R. G. da Silva, Elmar U. K. Melcher, Guido Araujo, and Valdiney Alves Pimenta. An automatic testbench generation tool for a systemc functional verification methodology. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 66–70, New York, NY, USA, 2004. ACM Press.

[34] David L. Dill. What's between simulation and formal verification? (extended abstract). In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 328–329, New York, NY, USA, 1998. ACM Press.

[35] Alfredo Ferro, Eugenio G. Omodeo, and Jacob T. Schwartz. Decision procedures for elementary sublanguages of set theory. i. multilevel syllogistic and some extensions. *Communications on Pure and Applied Mathematics*, 33(1):599–608, 1980.

[36] Alfredo Ferro, Eugenio G. Omodeo, and Jacob T. Schwartz. Decision procedures for some fragments of set theory. In *Proceedings of the 5th Conference on Automated Deduction*, pages 88–96, London, UK, 1980. Springer-Verlag.

[37] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 286–291, New York, NY, USA, 2003. ACM Press.

[38] Tony Fountain, Thomas Dietterich, and Bill Sudyka. Mining ic test data to optimize vlsi testing. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 18–25, New York, NY, USA, 2000. ACM.

[39] S. Furui. *Digital Speech Processing, Synthesis and Recognition, Second Edition*. CRC Press, 2000.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Padrões de projeto - soluções reutilizáveis de software orientado a objetos, 2000.

[41] JUDITH L. GERSTING. *Fundamentos Matemáticos para a Ciência da Computação*. LTC, Brasil, 2004.

[42] Mark Glasser, Adam Rose, Tom Fitzpatrick, Dave Rich, and Harry Foster. *Advanced Verification Methodology Cookbook*. Mentor Graphics, Cambridge, MA, USA, 2007.

[43] Alon Gluska. Practical methods in coverage-oriented verification of the merom microprocessor. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 332–337, New York, NY, USA, 2006. ACM.

[44] Raanan Grinwald, Eran Harel, Michael Orgad, Shmuel Ur, and Avi Ziv. User defined coverage - a tool supported methodology for design verification. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 158–163, New York, NY, USA, 1998. ACM Press.

[45] Aarti Gupta, Albert E. Casavant, Pranav Ashar, Akira Mukaiyama, Kazutoshi Wakabayashi, and X. G. (Sean) Liu. Property-specific testbench generation for guided simulation. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 524, Washington, DC, USA, 2002. IEEE Computer Society.

[46] O. Guzey and L.-C. Wang. Coverage-directed test generation through automatic constraint extraction. *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 151–158, 7-9 Nov. 2007.

[47] Michael Hahsler and Kurt Hornik. Building on the arules infrastructure for analyzing transaction data with r. In *Advances in Data Analysis, Proc. of the 30th Annual Conf. of the Gesellschaft fur Klassifikation e.V., Freie Universitat Berlin, March 8-10, 2006, Studies in Classification, Data Analysis, and Knowledge Organization*, pages 449–456. Springer-Verlag.

[48] Ian G. Harris. A coverage metric for the validation of interacting processes. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1019–1024, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[49] Richard C. Ho and Mark A. Horowitz. Validation coverage analysis for complex digital designs. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 146–151, Washington, DC, USA, 1996. IEEE Computer Society.

[50] C. Norris Ip. Simulation coverage enhancement using test stimulus transformation. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 127–134, Piscataway, NJ, USA, 2000. IEEE Press.

[51] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.

[52] V. Jerinić and D. Müller. Safe integration of parameterized ip. *Integr. VLSI J.*, 37(4):193–221, 2004.

[53] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, 2002.

[54] Michael Keating and Pierre Bricaud. *Reuse methodology manual: for system-on-a-chip designs*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[55] P. Maciel, E. Barros, and W. Rosenstiel. Estimating functional unit number in the PISH codesign system by using petri nets. In *Proceedings of the XII Symposyum on Integrated Circuits and Systems Design*, pages 32–35, 1999.

[56] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. S. Simao, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero. Proteum: a family of tools to support specification and program testing based on mutation. pages 113–116, 2001.

[57] N. MARRANGHELLO, W. L. A. OLIVEIRA, and F. DAMIANI. Exception handling with petri nets for digital systems. In *XV Simpósio Brasileiro de Concepção de Circuitos Integrados, 2002, Porto Alegre. Proceedings of the XV SBCCI*, pages 229–234, 2002.

[58] Andreas Meyer. *Principles of Functional Verification*. Newnes, 2003.

[59] Prabhat Mishra and Nikil Dutt. Functional coverage driven test generation for validation of pipelined processors. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 678–683, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Gabe Moretti, Tom Anderson, Janick Bergeron, Ashish Dixit, Peter Flake, Tim Hopes, and Ramesh Narayanaswamy. Your core— my problem? (panel session): integration and verification of ip. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 170–171, New York, NY, USA, 2001. ACM.

[61] Dinos Moundanos, Jacob A. Abraham, and Yatin V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Trans. Comput.*, 47(1):2–14, 1998.

[62] Sagar Naik. *Software Testing and Quality Assurance*. John Wiley & Sons, 2007.

[63] Zebo Peng. Synthesis of vlsi systems with the camad design aid. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 278–284, Piscataway, NJ, USA, 1986. IEEE Press.

[64] Isaac Maia Pessoa. Geração semi-automática de *testbenches* para circuitos integrados digitais. Master's thesis, 2007.

[65] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic, 2004.

[66] Bruno Otávio Piedade Prado. Ivm: Uma metodologia de verificação funcional interoperável, iterativa e incremental. Master's thesis, 2009.

[67] A. Randjic, N. Ostapcuk, I.Soldo, P. Markovic, and V. Mujkovic. Complex asics verification with systemc. In *23rd International Conference on Microelectronics*, pages 671–674, 2002.

[68] P. Rashinkar, P. Paterson, and L. Singh, editors. *System-on-a-Chip Verification Methodology and Techniques*. Kluwer Academic Publishers, Massachusetts, 2001.

[69] A. K. Rocha, P. Lira, E. Melcher Y. Y. Ju, and E. Barros. Silicon validated ip cores designed by the brazil-ip network. In *IP/SOC 2006 (IP-Based SoC Design)*, pages 142–147, 2006.

[70] C. L. Rodrigues, F. J. Morais, L. M. Silva, K. R. da Silva, D. D. S. Guerrero, J. C. A. de Figueiredo, and E. Melcher. Functional verification methodology using hierarchical coloured petri nets-based testbenches. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 103–110, 2008.

[71] Cássio L. Rodrigues, Karina R. G. da Silva, and Henrique N. Cunha. Improving functional verification of embedded systems using hierarchical composition and set theory. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1632–1636, New York, NY, USA, 2009. ACM.

[72] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on a multi-valued ar-automata. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 742–748, Piscataway, NJ, USA, 2001. IEEE Press.

[73] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multivalued ar-automata. In *Design, Automation and Test in Europe*, pages 742–748, 2001.

[74] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Checking temporal properties under simulation of executable system descriptions. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 161, Washington, DC, USA, 2000. IEEE Computer Society.

[75] C. Rust, A. Rettberg, and K. Gossens. From high-level Petri nets to SystemC. In *Systems, Man and Cybernetics, 2003. IEEE International Conference on , vol.2, 5-8 Oct.*, 2003.

[76] A. Samarah, A. Habibi, S. Tahar, and N. Kharma. Automated coverage directed test generation using a cell-based genetic algorithm. *High-Level Design, Validation, and Test Workshop, IEEE International*, 0:19–26, 2006.

[77] Youssef Serrestou and Vincent Beroulle Chantal Robach. Functional verification of rtl designs driven by mutation testing metrics. In *Proc. of the 10th Euromicro Conference on Digital System Design Architectures*, pages 222–227, Washington, DC, USA, 2007. IEEE Computer Society.

[78] George Sobral Silveira, Karina R. G. da Silva, and Elmar U. K. Melcher. Functional verification of an mpeg-4 decoder design using a random constrained movie generator. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 360–364, New York, NY, USA, 2007. ACM.

[79] Leena Singh, Leonard Drucker, and Neyaz Kahn. *Advanced Verification Techniques: a SystemC Based Approach for Sucessful Tapeout*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[80] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[81] Serdar Tasiran, Yuan Yu, and Brannon Batson. Linking simulation with formal verification at a higher level. *IEEE Des. Test*, 21(6):472–482, 2004.

[82] XviD Team. Xvid api 2.1 reference (for 0.9.x series). 2003.

[83] P. Vado, Y. Savaria, Y. Zoccarato, and C. Robach. A methodology for validating digital circuits with mutation testing. *IEEE Internat. Symp. on Circuits and Systems, 2000.*, 1:343–346 vol.1, 2000.

[84] A. Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[85] S. Vasudevan. *Effective Functional Verification Principles and Processes*. Springer, The Netherlands, 2006.

[86] Shireesh Verma, Kiran Ramineni, and Ian G. Harris. An efficient control-oriented coverage metric. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 317–322, New York, NY, USA, 2005. ACM.

[87] Ilya Wagner, Valeria Bertacco, and Todd Austin. Stresstest: an automatic approach to test generation via activity monitors. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 783–788, New York, NY, USA, 2005. ACM Press.

[88] Bruce Wile, John Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[89] Alex Yakovlev, Steve Furber, Krenz Krenz, and Alexandre Bystrov. Design and analysis of a self-timed duplex communication system. *IEEE Transactions on Computers*, 53(7):798–814, 2004.

[90] Praveen Yalagandula, Vigyan Singhal, and Adnan Aziz. Automatic lighthouse generation for directed state space search. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 237–242, New York, NY, USA, 2000. ACM Press.

[91] Steve Ye. Best practices for a reusable verification environment. In *Electronic Engineering Times (EE Times)*, New York, USA, 2004. http://www.eetimes.com.

[92] Jun Yuan, Jian Shen, Jacob A. Abraham, and Adnan Aziz. On combining formal and informal verification. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 376–387, London, UK, 1997. Springer-Verlag.

Apêndice A

Relatório Técnico: uma Extensão em Redes de Petri Coloridas da Metodologia de Verificação Funcional VeriSC

A.1 Introdução

O contexto geral deste trabalho é a engenharia de cirtuitos digitais integrados. Entendemos o termo engenharia como sendo a aplicação de um conjunto de conhecimentos cientificamente embasados para se produzir e manter artefatos que atendam às necessidades humanas. Os circuitos digitais integrados são Os sistemas de *hardware* atuais são constituídos de diversos componentes, chamados *IPs cores* (*Intellectual Property core*) que operam sobre um único circuito integrado (*chip*). Antes que os IPs sejam integrados em um único *chip*, é fundamental que eles sejam validados para que erros sejam eliminados antes da transformação do sistema em uma pastilha de silício. Neste caso, validar significa certificar-se de que cada IP respeita sua especificação.

A técnica amplamente empregada para validação de *hardware* é a verificação funcional [18; 88]. A verificação funcional consiste em comparar o *design* do IP, em linguagem de descrição de *hardware*, de agora em diante designado por DUV (*Design Under Verification*), com um modelo de mais alto nível de abstração, que exerce papel de especificação, de agora

em diante designado por Modelo de Referência [1]. Comparar os dois modelos consiste em fornecer o mesmo conjunto de estímulos para os dois modelos e avaliar se as respostas obtidas são iguais. A discrepância de resultados significa que DUV em desenvolvimento possui erro.

Essa comparação de dois modelos mencionada no parágrafo anterior, contudo, é um processo que envolve outros elementos além desses dois modelos. Um elemento chave na verificação funcional é o *ambiente de verificação funcional*, também chamado de *testbench*. O *testbench* é o ambiente que interage com o DUV. Além do modelo de referência, o *testbench* é responsável por abarcar os elementos que são responsáveis pela geração de estímulos e pela comparação automática dos resultados de saída.

A qualidade da verificação funcional depende de mecanismos para se certificar que todas as funcionalidades do IP foram exercitadas. O mecanismo empregado pelos engenheiros para se fazer essa certificação se chama análise de cobertura. Na análise de cobertura, o engenheiro especifica um conjunto de metas que devem ser atingidas durante a verificação funcional. Uma vez que estas metas são atingidas, o processo de verificação funcional pode ser encerrado e uma nova etapa na construção do *hardware* pode ser iniciada. Assim, a verificação funcional não garante que o IP está isento de erros, ela garante que as funcionalidades especificadas foram exercitadas.

A validação é uma atividade crítica no desenvolvimento de um *hardware*, consumindo em média cerca de 70% dos recursos do projeto. O alto custo desta atividade exige que a verificação funcional seja considerada com rigor desde as fases iniciais do desenvolvimento do sistema [18; 88]. VeriSC [31; 32] é uma metodologia de verificação funcional que permite que a validação seja considerada desde as fases iniciais do desenvolvimento do *hardware*. VeriSC determina que o DUV e o ambiente de verificação sejam desenvolvidos a partir de uma especificação única para minimizar os *custos de integração* de ambos na verificação. VeriSC também determina que o ambiente de verificação seja construído antes do DUV para que o DUV possa ser validado no ambiente de verificação desde o início de sua concepção. O fluxo de verificação funcional determinado por VeriSC, com o suporte ferramental para a geração do ambiente de verificação, permitiu a redução de 30% [31; 32] do tempo de validação do decodificador MPEG 4[69] em relação a uma metodologia

[1] em inglês, este modelo também designado por *golden model*

tradicional de verificação funcional, que também possui suporte ferramental para a geração do ambiente de verificação, mas desenvolve primeiro o DUV para depois desenvolver o ambiente de verificação, sem o uso de especificação única para o desenvolvimento de ambos.

VeriSC, a exemplo de outras metodologias de verificação existentes [24; 67; 68; 87], emprega variações de uma mesma linguagem para descrição do DUV e dos elementos do ambiente de verificação funcional. No projeto MPEG 4, por exemplo, VeriSC foi empregada com Modelo de Referência descrito em C-C++, elementos do ambiente de verificação descritos em SystemC [7] comportamental, isto é, com elementos de mais alto nível de abstração, e DUV em linguagem de descrição de *hardware*. Embora SystemC seja adequada para a descrição do DUV, o uso de SystemC e C-C++ nos demais elementos do ambiente de verificação funcional não permite a realização de Verificação formal do Modelo de Referência: em um Modelo de Referência em C-C++, não é possível a aplicação efetiva de técnicas formais para assegurar que ele respeita às especificações do sistema. Como conseqüência, um erro de implementação no Modelo de Referência pode indicar de maneira equivocada um erro do DUV. Este tipo de problema aconteceu na aplicação de VeriSC no projeto MPEG 4. O Modelo de Referência usado foi o XVID [2]. Este modelo, em C, apresenta um problema no bloco chamado *Bitstream*. O papel deste bloco é receber um fluxo de vídeo compactado e encaminhar as informações necessárias para que cada bloco possa executar seu serviço.

Uma solução para os problemas citados anteriormente é o uso de linguagens de modelagem formais para a descrição do ambiente de verificação. Redes de Petri coloridas [51], por exemplo, é um formalismo com nível de abstração mais alto que o de linguagens de descrição de *hardware*, que pode ser usado para a descrição do ambiente de verificação. Redes de Petri coloridas podem contribuir com a qualidade do Modelo de Referência, pois as Redes de Petri coloridas são verificáveis e muitas das técnicas de verificação podem ser realizadas automaticamente [4]. Assim, o engenheiro de verificação tem como obter garantias de que o Modelo de Referência está obedecendo às especificações que o IP deve respeitar.

A.1.1 Objetivos

O objetivo deste trabalho é *estender* a metodologia de verificação funcional VeriSC para que ela funcione com ambientes de verificação funcional em redes de Petri coloridas. Este objetivo maior pode ser decomposto nos seguintes objetivos específicos:

- Definir um ambiente de verificação funcional em redes de Petri coloridas com suporte ferramental para a construção do mesmo.
- Desenvolver a análise de cobertura de valores entrada e saída.
- Contruir suporte ferramental suficiente para a análise de cobertura.
- Realizar estudo comparativo entre o trabalho proposto e a metodologia VerisC original.

A.1.2 Justificativa e Relevância

A crescente complexidade dos sistemas de hardware e a necessidade de que o tempo de produção destes sistemas satisfaça às necessidades de mercado exigem um aprimoramento constante de técnicas de validação para se garantir um sistema livre de erros. Uma maneira de se alcançar tal aprimoramento é mediante a combinação de diversas técnicas de modo a preservar o que há de mais positivo em cada uma delas [20; 34]. Segundo Bhadra *et al* [20], estas abordagens que combinam mais de uma técnica de validação são chamadas de *abordagens híbridas*. Ainda segundo Bhadra07 *et al*, as referência [6; 3; 5] são exemplos de abordagens híbridas desenvolvidas pela indústria. Vários outros pesquisadores relatam trabalhos de abordagens híbridas como forma de se aperfeiçoar a atividade de validação [81; 49; 61; 92; 73; 90]

Dill [34] também sugere que forma de se conseguir aperfeiçoamento da validação é mediante a combinação de múltiplas técnicas. De forma específica, Dill enfatiza a combinação de técnicas formais com técnicas baseadas em simulação. O trabalho que propomos neste documento é uma combinação de técnicas formais com técnicas baseadas em simulação. A contribuição consiste no aperfeiçoamento da verificação funcional com o uso de redes de Petri coloridas. Redes de Petri, de uma forma geral, é um formalismo interessantes para a modelagem e validação de sistemas de *hardware* [21; 23; 55; 57; 75; 89; 63]. Questões a respeito de *sincronização*, *compartilhamento de recursos* e *comunicação* podem ser abordados de maneira rigorosa. Assim, depois da conclusão deste trabalho, será possível realizar a verificação funcional com um ambiente de verificação formal. Na prática, isto significa que o engenheiro de verificação terá um mecanismo a mais para se assegurar que*design* do sistema respeita suas especificações, podendo contribuir, portanto, com a

qualidade do *design* e com a redução tempo de produção do mesmo.

A.2 Fundamentação Teórica

Nesta Seção, apresentamos os conceitos básicos para a compreensão do trabalho proposto. Três tópicos básicos são abordados: verificação funcional, análise de cobertura e redes de Petri coloridas.

Embora a verificação funcional seja amplamente empregada na indústria, não existe uma metodologia padrão, que seja adequada a todos os tipos de sistemas. Logo, cada indústria pode possuir sua metodologia de verificação funcional, a depender do tipo de sistema a ser desenvolvido e das ferramentas disponíveis para automação de suas atividades. O mesmo também é verdade a respeito da análise de cobertura. O consenso que existe a respeito de cobertura é relacionado aos tipos de análises existentes. Os tipos de análise empregados e suas implementações vão variar de aplicação para aplicação. Por estes motivos, neste capítulo, primeiro apresentamos uma discussão básica sobre verificação funcional e análise de cobertura, nas seções A.2.1 e A.2.1 respectivamente, para em seguida,no Capítulo A.4.10, apresentar VeriSC, a metodologia de verificação funcional a ser usada no trabalho proposto. A apresentação de VeriSC está em capítulo separado para simplificar o entendimento do leitor do impacto causado pelo usado de redes de Petri colorida na mesma, conforme apresentado no Capítulo A.4.10. Mais detalhes sobre os tópicos principais deste capítulo podem ser encontrados em [18; 31; 32; 51]

A.2.1 Verificação Funcional

A verificação funcional é uma técnica utilizada para verificar se o IP em desenvolvimento respeita suas especificações. A idéia básica desta técnica consiste em *comparar* o design do dispositivo a ser desenvolvido, de agora em diante DUV (Design Under Verification), com um modelo de referência. Esse modelo de referência, por definição, respeita as especificações que o DUV também deve respeitar. A comparação consiste em fornecer os mesmos estímulos para os dois modelos e comparar os resultados produzidos por cada um deles. A ocorrência de valores de saída diferentes significa que o DUV possui erro. A verificação

funcional não prescreve quais linguagens devem ser usadas para definição dos modelos. No entanto, tradicionalmente, o DUV é descrito em uma linguagem de descrição de hardware, tal como Verilog [1], e o modelo de referência é descrito em C/C++ ou SystemC [18].

Essa comparação de dois modelos mencionada no parágrafo anterior, contudo, é um processo que envolve outros elementos além desses dois modelos. Um elemento chave na verificação funcional é o ambiente de verificação funcional, também chamado de *testbench*. O *testbench* é o ambiente que envolve o DUV. Ele é responsável por abarcar os elementos que são responsáveis pela geração de estímulos e pela comparação automática dos resultados de saída.

O ambiente de verificação deve ser implementado preferencialmente em um nível de abstração alto, denominado de nível de transação (*Transaction-level*). Esse nível de transação não se preocupa com detalhes de protocolos no nível de sinais. Ao invés disso, o seu foco é a comunicação entre blocos e a transferência de transações. Uma transação é uma representação básica para a troca de informações entre dois blocos funcionais. Em outras palavras, é uma operação que inicia num determinado momento no tempo e termina em outro, sendo caracterizada pelo conjunto de instruções e dados necessários para realizar a operação. Ela é definida pelo seu tempo inicial, tempo final e atributos [24]. Um exemplo de uma transação poderia ser uma transmissão de um pacote *ethernet*.

Análise de Cobertura

Um aspecto crítico da verificação funcional é a detecção de seu término. Idealmente, a verificação funcional deve terminar assim que todas as funcionalidades implementadas tenham sido exercitadas. Porém, esta constatação depende da qualidade e da quantidade de estímulos utilizados, pois a comparação dos resultados do modelo de referência e DUV são realizadas através de simulação. Assim sendo, é muito importante que exista um mecanismo para detectar se todas as funcionalidades especificadas foram exercitadas.

Para responder se todas as funcionalidades especificadas foram exercitadas, os engenheiros de verificação usam análise de cobertura. A análise de cobertura é uma técnica usada para medir o progresso da verificação e reportar quais funcionalidades deixaram de ser exercitadas. A análise de cobertura pode ser compreendida como sendo um conjunto de metas que devem ser atingidas durante a verificação funcional. Essas metas podem ser especifica-

das em função de diversos critérios. Segundo esses critérios, podemos classificar dois tipos de cobertura principais: análise de cobertura de código e análise de cobertura funcional.

Análise de Cobertura de Código

Para a cobertura de código, a ferramenta de análise de cobertura vai reportar quais partes de código foram executadas e quais não foram durante a simulação. Este tipo de análise é importante por revelar ao engenheiro se existe alguma parte do design que não foi exercitada. A existência de partes do design que não foram exercitadas é ruim porque elas podem conter algum erro. Este tipo de cobertura requer algum tipo de instrumentação do código. Esta instrumentação consiste de pontos de observação no código para registrar se tal parte foi exercitada de fato. As seguintes métricas podem ser usadas para este tipo de cobertura: linhas de código, caminhos de execução e expressões.

Na cobertura de linhas de código, a ferramenta mede quais *linhas* exatamente foram executadas e quais não foram. Para se alcançar 100 $\%$ de cobertura de código é necessário compreender quais condições lógicas devem ser satisfeitas para se alcançar as linhas descobertas. Contudo, é muito comum que a codificação defensiva leve a produção de blocos de comandos que nunca são executados, fazendo com que a cobertura total nunca seja alcançada.

A cobertura de caminhos de execução mede as possíveis *seqüências* de linha de comando que podem ser executadas em um design. A existência de comandos de fluxo de controle, tais como estruturas condicionais do tipo *if then else*, faz com que existam vários caminhos de execução. Por exemplo, temos um caminho em que o bloco *then* é executado e temos outro referente ao bloco *else*. Este tipo de cobertura é bem mais precisa que a cobertura de linhas de código, pois um erro pode ser revelado somente quando uma seqüência específica ocorre. Em contra-partida, o número de seqüências cresce exponencialmente em função do número de comandos de fuxo de controle.

Ainda mais precisa que a cobertura de caminhos de execução, a cobertura de expressões analisa as diversas *instâncias* que um caminho de execução pode ocorrer. Por exemplo, a condição de um bloco *if* pode conter uma expressão com o operador lógico *ou*. Naturalmente, esta expressão pode ser satisfeita quando um dos operandos for *verdadeiro*, fazendo com que exista pelo menos três instâncias do mesmo caminho de execução.

Análise de Cobertura Funcional

Se a análise de cobertura de código mede o quanto do código foi exercitado, a análise de cobertura funcional mede o quanto da *especificação* original foi exercitada. Ela pode analisar, por exemplo, o nível de ocupação de um *buffer*, a quantidade de pacotes enviados, requisição de barramento etc. Assim, a cobertura funcional está focada no *propósito* da função implementada enquanto a cobertura de código está focada na execução do código.

A cobertura de código, portanto, depende do domínio da aplicação. Na prática, isto significa que a especificação dos critérios de cobertura deve ser realizada manualmente pelo engenheiro de verificação, isto é, os critérios de cobertura não podem ser extraídos automaticamente do código em linguagem de descrição do *hardware*. Assim como na análise de cobertura de código, os valores das execuções são extraídos durante a simulação e armazenados em uma base de dados. A partir dessa base, a análise de cobertura ocorre propriamente em função do que foi especificado. Os critérios comumente utilizados para cobertura funcional são os seguintes: cobertura de valores escalares individuais e cobertura cruzada.

Na cobertura de valores escalares, o engenheiro especifica o conjunto de valores relevantes que devem ser observados na verificação, seja como estímulos de entrada, seja como resultados de saída. Exemplos de valores escalares utilizados para este tipo de cobertura são: tamanho de pacote, ocupação de *buffer*, acesso a barramento etc. É uma tarefa muito simples especificar e medir cobertura desta natureza, sendo que as medições chegam bem perto de 100 % na maioria das vezes. É importante que fique claro que 100% de cobertura não garante que o *design* está isento de erros. Isto que significa que todos os critérios especificados foram totalmente satisfeitos.

A cobertura cruzada de valores [2] trata de medir a ocorrência da *combinação* de diversos valores. Ela é útil para especificar propriedades do tipo: " Um pacote corrompido foi inserido em todas as portas?" "Todos os *buffers* ficaram preenchidos ao mesmo tempo?". A implementação de cobertura cruzada segue o mesmo princípio da cobertura de valores escalares, a diferença é que na cobertura cruzada várias valores são coletados ao mesmo tempo.

[2] *cross coverage*, em inglês

A.2.2 Redes de Petri Coloridas Hierárquicas

Apresentação Informal

As redes de Petri coloridas (CPN[3]) são um formalismo gráfico-matemático para modelagem, validação e verificação de sistemas de software e hardware. Uma rede CPN é grafo bipartido cujos elementos são chamados de lugares e transições, graficamente representados por elipses e retângulos respectivamente. Os lugares servem para armazenar os dados do sistema modelado e as transições modelam a ocorrência dos eventos. A ocorrência de eventos em qualquer sistema requer que certas condições sejam satisfeitas e a ocorrência destes eventos também pode alterar o estado do sistema. Para lidar com estas condições e com as alterações dos estados do sistema, os lugares são ligados a transições e as transições a lugares através de arcos. Para expressar exatamente estas condições e como o sistema deve ficar após a ocorrência do evento, cada arco possui uma expressão associada a ele.

Para apresentar os elementos de uma rede de Petri com mais detalhes, vamos considerar o procedimento do DPCM modelado em CPN conforme ilustrado na Figura A.1

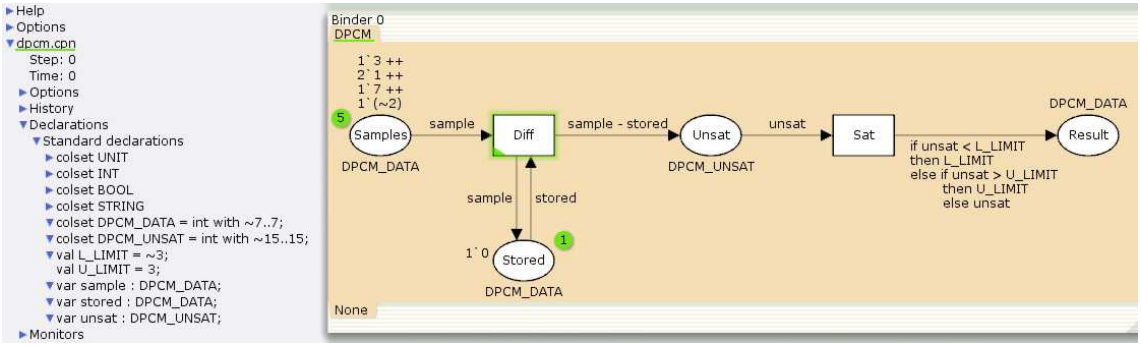

Figura A.1: Rede de Petri colorida do modelo DPCM.

Cada lugar da rede de Petri possui um tipo de dado, também chamado de cor, que especifica o conjunto de valores que ele é capaz de armazenar. Este tipo pode ser um tipo básico, tais como um inteiro ou um valor *booleano*, ou pode ser um tipo composto definido pelo o usuário a partir de outros tipos básicos, tal como um tipo *pacote*, composto por identificadores de rementente, destinatário e conteúdo. No caso do DPCM da Figura A.1, os dados de entrada ficam armazenados inicialmente no lugar *Samples*. O tipo desse lugar é *DPCM_DATA*. O processamento do dado de entrada é modelado pela transição *Diff*. Os

[3]Coloured Petri Nets, em inglês.

arcos que chegam nesta transição representam a condição que deve ser satisfeita para que o processamento ocorra, ou seja, deve existir um valor na entrada, armazenado no lugar *Samples* e um valor previamente lido, armazenado no lugar *Stored*. Quando a transição *Diff* ocorre, os valores dos lugares *Sample* e *Stored* são removidos. De modo atômico, o valor lido de *Samples* é armazenado em *Stored* e o resultado da subtração é armazenado no lugar *Unsat*. Quando o valor da subtração é armazenado em *Unsat*, a transição *Sat* pode ocorrer, isto é, a rede reune as condições necessárias para que a saturação ocorra. A saturação de fato está modelada pelo arco que liga *Sat* ao lugar *Result*. A expressão associado a este arco se encarrega de testar se o valor lido está dentro faixa ou não e fazer a devida saturação.

O procedimento do DPCM é simples de modo que o tamanho da rede de Petri em termos dos elementos que a compõe não afeta o entendimento do modelo e sua análise. Mas sistemas de hardware são mais complexos que o DPCM e a modelagem desses sistemas em uma única rede, sem qualquer mecanismo de (de)composição, se torna inviável na prática. Um formalismo que permite que modelos complexos sejam compostos de vários modelos menores são as redes de Petri coloridas hierárquicas (HCPN[4]).

As redes de Petri coloridas hierárquicas permitem a decomposição do modelo através das transições de substituição. Uma transição de substituição representa uma outra rede de Petri, ou seja, ela abstrai a ocorrência de um evento mais complexo. Um exemplo de modelo em HCPN é apresentado na Figura A.2. Este exemplo é uma modelagem hierárquica do DPCM da Figura A.1. A transição *DIF* modela a operação de diferença de amostras de entrada do DPCM. O resultado da diferença é armazenado no lugar *Unsat*. Esta diferença passará pelo processo de saturação na transição de substituição *SAT*. Os conteúdos das páginas *DIF* e *SAT* estão ilustrados nas figuras A.3 e A.4. A rede Figura A.2 é chamada de superpágina e as páginas *DIF* e *SAT* são as subpáginas. O lugar *Samples* na rede *DIF* é o lugar através do qual ocorre a comunicação com a página DPCM. O lugar de comunicação da superpágina é chamado de *socket* e na subpágina eles é chamado de *porta*.

Apresentação Formal

Definição A.1 (Estrutura de redes de Petri) *Uma estrutura de rede de Petri é uma tupla* $\langle P, T, F \rangle$*, em que* P *é um conjunto finito de* lugares*,* T *é um conjunto finito de* transições *e*

[4] *Hierarchical Coloured Petri Nets*, em inglês

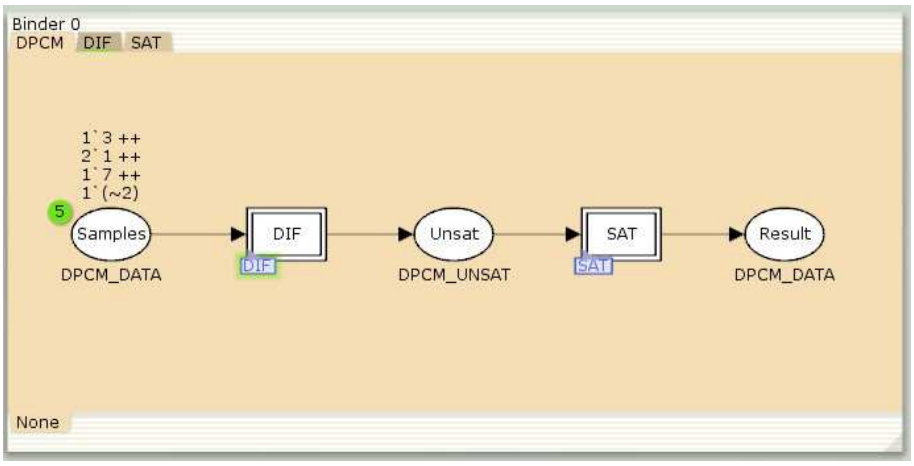

Figura A.2: Rede de Petri colorida hierárquica do modelo DPCM.

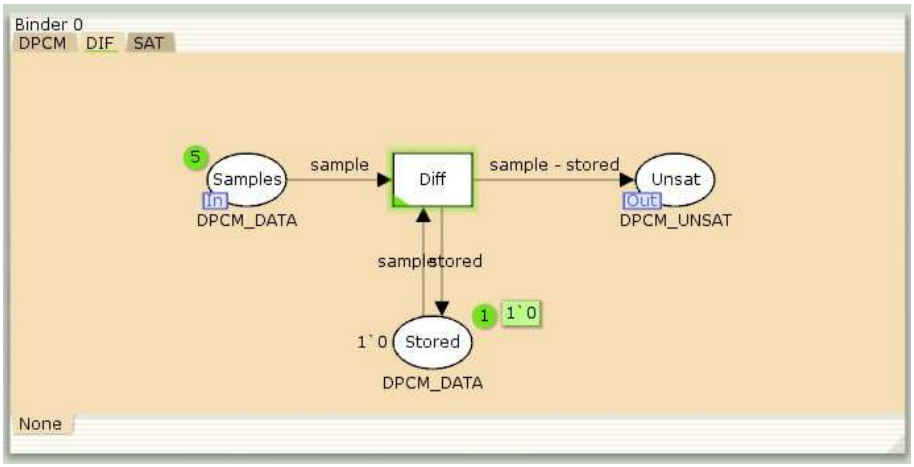

Figura A.3: Detalhamento da transição de substituição DIF.

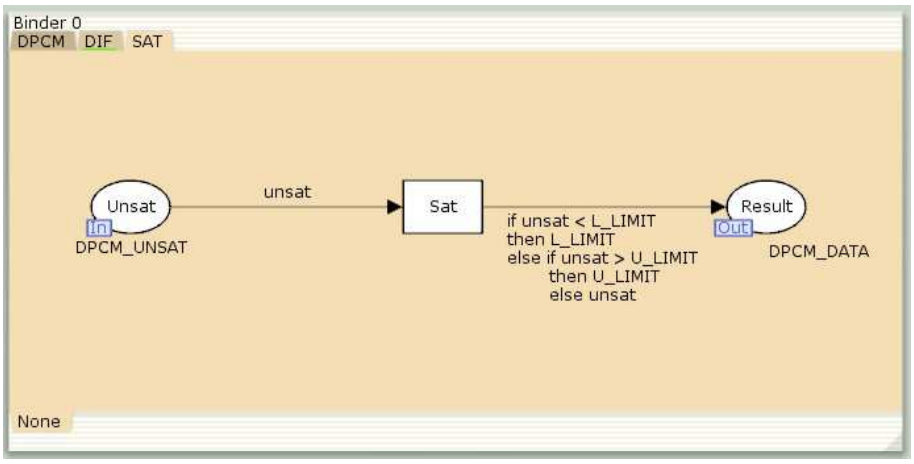

Figura A.4: Detalhamento da transição de substituição SAT.

$F \subseteq (P \times T) \cup (T \times P)$ *é um conjunto de* arcos*, com* P *e* T *disjuntos.*

Além de uma estrutura, uma rede de Petri colorida possui declarações de tipos de lugares, expressões de arcos, guardas nas transições, e expressões de inicialização de marcação. Todas estas definições são construídas a partir de elementos da assinatura de Σ.

Definição A.2 (Rede de Petri colorida) *Sejam uma assinatura* Σ*, uma estrutura de rede de Petri* $N = \langle P, T, F \rangle$ *e funções* $C : P \to \mathbb{S}$ *,* $G : T \to 2^{\mathcal{E}}$*,* $E : F \to \mathfrak{T}^{ms}$*, e* $I : P \to \mathfrak{T}^{ms}$ *denominadas, respectivamente, de* função de cores*,* de guardas*,* de expressões de arcos*, e de* inicialização*. Dizemos que* $\langle N, C, G, E, I \rangle$ *é uma* rede de Petri colorida *definida sobre* Σ *desde que:*

$$\forall f \in F : \; E(f) \text{ seja do tipo } C(p), \text{ onde } f = \langle p, t \rangle \text{ ou } f = \langle t, p \rangle \text{ e}$$

$$\forall p \in P : \; I(p) \text{ é uma expressão fechada (não tem variáveis) e é do tipo } C(p).$$

A função de cores, que mapeia os lugares da rede a sortes, tem por finalidade determinar o tipo de dado correspondente a cada lugar. Os elementos de um domínio existentes em cada lugar da rede são denominados *fichas*. A função de guarda mapeia equações a transições. A função de expressão de arcos indica o "peso" do arco, ou seja, a quantidade de fichas a serem retiradas ou inseridas nos lugares associados ao arco. Para que uma transição seja habilitada, as expressões da função de guarda e da função de expressões devem ser satisfeitas. Por último, a função de inicialização associa uma expressão fechada a cada lugar para determinar sua marcação inicial.

Comportamento das Redes de Petri Com os elementos apresentados até agora, nós podemos definir o que é *marcação* e o que *estado* de uma rede de Petri. Chamamos *marcação* o conjunto formado pela relação *lugares da rede X fichas*. A marcação determina o *estado* de uma rede de Petri.

Definição A.3 (Fichas e Marcações) *Uma* ficha *é um par* $\langle p, d \rangle \in P \times \mathsf{D}_i$*, tal que* $C(p) = \mathbb{S}_i$*. Denotamos por* F *o conjunto das fichas de uma rede. E uma* marcação *é um multiconjunto sobre* F*. O conjunto das marcações de uma rede é denotado por* M*.*

A marcação de uma rede de Petri somente se modifica mediante o *disparo* (ou *ocorrência*) de uma transição. O disparo das transições, por sua vez, determina o comportamento

da rede. As expressões dos arcos e as fichas contidas nos lugares de entrada de uma transição desempenham o papel de *pré-condições* para que o disparo de uma transição ocorra. A atribuição de valores às variáveis contidas nas expressões de arcos e guardas associadas às transições determina o *modo* de disparo para uma transição. Naturalmente, vários modos de disparo podem existir para uma transição.

Definição A.4 (Modo de transição) *Um* modo *de uma transição t é uma atribuição de valores para as variáveis de $\mathcal{V}$. Se denotamos a atribuição por a, então escrevemos t^a para denotar a transição t no modo a.*

Quando conveniente, subentendemos a e escrevemos apenas t. Este recurso deve ser utilizado para simplificar a notação e enfatizar a transição em detrimento do modo. A partir da definição de modo, podemos determinar a regra de disparo de uma transição.

Definição A.5 (Transição habilitada) *Sejam t uma transição, m uma marcação e a um modo de t. Dizemos que t^a está* habilitada *na marcação m, e denotamos isso por $m[t^a\rangle$, se $[\![e]\!]_a$ é verdade para toda equação $e \in G(t)$ e se*

$$[\![E(p,t)]\!]_a \leq m(p) \quad \textit{para todo } p \in P.$$

Desta forma, duas condições definem a habilitação e disparo de uma transição. Uma é que a expressão da guarda seja avaliada para verdadeira no modo a. A outra condição é que existam mais fichas nos lugares de entrada da transição do que o resultado da avaliação do arco de entrada da transição. Disparar uma transição pode mudar o estado de uma rede. A seguir, definimos a marcação alcançada após o disparo.

Definição A.6 (Marcação alcançável) *Sejam t^a e m tais que $m[t^a\rangle$. Temos que m' é a marcação alcançada a partir da ocorrência de t^a na marcação m. Denotamos $m[t^a\rangle m'$ e definimos m' por*

$$m'(p) = m(p) - [\![E(p,t)]\!]_a + [\![E(t,p)]\!]_a \qquad \textit{para todo } p \in P. \tag{A.1}$$

Se $m[t^a\rangle m'$, dizemos que m' é diretamente alcançável *a partir de m. Chamamos cada $m[t^a\rangle m'$ de* disparo *ou de* ocorrência*.*

A definição acima indica que fichas são retiradas dos lugares de entrada e inseridas nos lugares de saída de t de acordo com a avaliação das expressões dos arcos que ligam t aos seus lugares de entrada e saída.

Definição A.7 (Rede de Petri colorida hierárquica) *Uma rede de Petri colorida hierárquica é uma tupla HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP) satisfazendo os seguintes requisitos:*

1. *S é um conjunto finito de páginas tal que:*
   - *Cada página* $s \in S$ *is é uma rede de Petri não hierárquica:* $(\Sigma_s, P_s, T_s, A_s, N_s, C_s, G_s, E_s, I_s)$.
   - *O conjunto de elementos são disjuntos par a par:* $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (Ps_1 \cup Ts_1 \cup As_1) \cap (Ps_1 \cup Ts_1 \cup As_1) = \emptyset]$
2. *SN* $\subseteq$ *T é um conjunto de nós de substituição.*
3. *SA é uma função de associação de páginas. SA é definida de SN para S tal que:*
   - $\{s_0 s_1 ... s_n \in S * | n \in \mathcal{N}_+ \wedge s_0 = s_n \wedge \forall k \in SA(SNs_{k-1})\} = \emptyset$
4. *PN* $\subseteq$ *P é um conjunto de nós portas.*
5. *PT é função de tipo de porta. Ela é definida de PN em {in, out, io, general}.*
6. *PA é uma função de associação de porta. Ela é definida de SN para relações binárias tais que:*
   - *Nós* sockets *são associados com nós portas:* $\forall t \in SN : PA(t) \subseteq X(t) \times PN_{SA(t)}$.
   - *Nós* sockets *são do tipo correto:* $\forall t \in SN \; \forall (p1, p2) \in PA(t) : [PT(p2) \neq general \Rightarrow ST(p1, t) = PT(p2)]$.
   - *Nós relacionados têm tipos iguais e a mesma expressão inicial:* $\forall t \in SN \forall (p1, p2) \in PA(t) : [C(p1) = C(p2)] \wedge I(p1)_{<>} = I(p2)_{<>}]$.
7. *FS* $\subseteq P_S$ *é um conjunto finito de conjuntos de fusão tais que:*

- *Membros de um conjunto de fusão têm o mesmo tipo e a mesma expressão inicial:* $\forall fs \in FS : \forall p1, p2 \in fs : [C(p1) = C(p2) \wedge I(p1)_{<>} = I(p2) <>]$.
- *FT é uma função de tipo de fusão. Ela é definida de conjuntos de fusão em {global, page, instance} tal que: conjuntos de página e de instâncias de fusão pertence a mesma página:* $\forall fs \in FS : [FT(fs) \neq global \Rightarrow \exists s \in S : fs \subseteq Ps]$
- *PP* $\in$ *S é um multi-conjunto de páginas primárias.*

A.3 Trabalhos relacionados

Em [44], Grinwald *et al* apresentam uma metodologia de cobertura funcional para *design* RTL que separa o modelo de cobertura da análise de cobertura propriamente. Esta separação habilita o engenheiro a definir modelos de coberturas específicos para o circuito digital em desenvolvimento. O suporte ferramental desta metodologia é a ferramenta *Comet (COverage MEtric Tool)*. O engenho base desta ferramenta é um banco de dados relacional que dá suporte ao armazenamento do dados da verificação funcional, análise de cobertura e geração de relatórios. Segundo Grinwald, a metodologia e a ferramenta foram usadas em diferentes projetos da IBM.

Em relação à analise de cobertura existente em VeriSC e ao trabalho do que está sendo proposto, o trabalho de Grinwald *et al* se diferencia principalmente no artefato em que ocorre a análise de cobertura. Enquanto o VeriSC é uma metodologia *black box*, isto é, a análise de cobertura ocorre somente nas interfaces do ambiente de verificação e no modelo de referência, o trabalho de Grinwald *et al* é *white box*, isto é, permite que a análise de cobertura seja realizada no *design* RTL de fato. Em VeriSC a análise de cobertura *pode* até ser realizada no *design* RTL, porém isto só será possível depois que todo o ambiente de verificação for construído. Mas nada impede, por exemplo, que o trabalho de Grinwald *et al* e o trabalho que está sendo proposto sejam complementares. Primeiro, a verificação funcional é realizada conforme VeriSC. Caso exista algum critério de cobertura que pode ser especificado somente sobre o *design* RTL, faz-se a aplicação do trabalho de Grinwald *et al*.

Abarbanel *et al*, em [12], apresentam um trabalho sobre a geração automática critérios de cobertura a partir de especificação formal. A especificação formal é a mesma utilizada para a

técnica de verificação de modelos. Portanto, Abarbanel *et al* assumem uma metodologia que faz uso de verificação de modelos[27] e verificação funcional. Em resumo, uma ferramenta traduz a especificação formal em um *checker*. Este *checker* é combinado com o *design* e simulado com o mesmo durante a verificação funcional. A especificação formal usada para a análise de cobertura contempla um subconjunto de operadores CTL, de modo que cada fórmula resultante é equivalente a uma asserção. Abarbanel *et al* relatam redução de até 50 % na verificação como um todo.

O trabalho proposto neste documento objetiva uma análise de cobertura com especificação mais complexa que asserções, tal como apresentado por Abarbanel. A especificação de caminhos de execução engloba múltiplos eventos do *design* de alto nível através de uma relação de ordem entre os mesmos. A idéia de integrar a verificação de modelos e a verificação funcional através das especificações formais deveria ser considerada nesta proposta. Porém, por limitação de tempo e recursos, esta possibilidade não será investigada neste trabalho.

Em [74; 72], Ruf *et al* apresentam uma técnica de verificação de propriedades em lógica temporal usando simulação. O trabalho foi desenvolvido para SystemC. Em resumo, o *design* RTL é instrumentado com propriedades que devem ser respeitadas durante a simulação. Estas propriedas são convertidas em autômatos AR, (*Accepting-Rejecting*) que possuem estados de aceitação e rejeição. Durante a simulação, os eventos que ocorrem no *design* são processados no autômato. Caso o autômato termine o processamento em um estado de rejeição, a ferramenta reporta este processamento como erro para o engenheiro.

A idéia básica do trabalho de Ruf *et al* pode ser interpretado como uma abordagem complementar ao trabalho proposto neste documento pois trabalhos se apoiam em artefatos diferentes. VeriSC é uma metodologia de verificação funcional cuja especificação a ser respeitada é definida pelo Modelo de Referência. No trabalho de Ruf *et al*, a especificação a ser respeitada é definida pela fórmula em lógica temporal. Logo, os dois trabalhos podem produzir resultados diferentes, aumentando a confiança no *design* produzido.

Em [21], uma abordagem para verificação de controladores paralelos usando redes de Petri de baixo nível é apresentada. A idéia deste trabalho é construir um modelo de referência em redes de Petri no mesmo de nível de abstração do RTL. Do modelo em redes de Petri é extraído o grafo de ocorrência. Este grafo de ocorrência é utilizado para comparação com o *design* RTL. Os autores da referência [21] não discutem o problema da explosão de espaço

de estados e não dão detalhes a respeito da verificação de outras aplicações.

A.4 A Metodologia VeriSC usando Redes de Petri Coloridas

VeriSC foi concebida originalmente com o ambiente de verificação em C-C++ e SystemC. O uso de redes de Petri coloridas depende da adequação da metodologia VeriSC para que ela suporte ambiente de verificação em redes de Petri coloridas. Esta adequação deve preservar as características originais da metodologia VeriSC e permitir que os benefícios do uso redes de Petri coloridas sejam incorporados (vide Capítulo 1 para informações sobre tais benefícios). Vários critérios podem ser usados para mensurar essa preservação. Por limitação de tempo e de recursos humanos, na discussão da adequação serão considerados a preservação de fluxo de atividades, o suporte ferramental e o domínio de aplicação.

Para apresentar a adequação, as duas metodologias serão apresentadas lado a lado ao longo da seções seguintes. Para facilitar a apresentação, será utilizado o exemplo do DPCM, apresentado no Capítulo A.2. Por questões de simplificação, a metodologia VeriSC sem redes de Petri será referenciada pelo termo VeriSC e a metodologia de verificação funcional com redes de Petri será referenciada por VeriSC-RP. É importante destacar que o texto que apresenta VeriSC [31] usa o termo *testbench* para referenciar o ambiente de verificação. Logo, deve ficar claro para o leitor que *testbench* e ambiente de verificação têm o mesmo significado ao longo deste documento.

A.4.1 Propriedades do ambiente de verificação

VeriSC

Os elementos que compõem o ambiente de verificação da metodololologia são aqueles que estão inseridos na área cinza da Figura A.5. Os blocos existentes são *Source, TDriver, TMonitor*, Modelo de Referência e *Checker*. A comunição entre os blocos, feita usando-se uma estrutura do tipo FIFO (*First In First Out*), está representada por setas mais largas. O ambiente de verificação se comunica com o DUV por interfaces formadas por sinais, ilustradas pelas setas mais finas da figura.

O ambiente de verificação de VeriSC possui quatro características básicas:

- É dirigido por coberturas. A verificação funcional pára somente quando os critérios de cobertura são satisfeitos. Logo, os estímulos devem ser gerados visando a progressão da análise de cobertura.

- Suporte à geração aleatória de estímulos. VeriSC ainda não tem suporte ferramental para auxiliar na geração de estímulos que satisfaçam diretamente a cobertura especificada. O suporte ferramental que existe para auxiliar na geração de estímulos é o uso de funções aleatórias. Através destas funções aleatórias que os estímulos são gerados até que os critérios de coberturam sejam satisfeitos. A escolha dessa funções tem impacto na qualidade dos estímulos gerados e depende do conhecimento do engenheiro a respeito do *design* a ser verificado.

- É autoverificável. A quantidade de estímulos usados para atingir critérios de cobertura é grande o suficiente para tornar a comparação manual dos resultados produzidos pelo Modelo de Referência e pelo DUV uma atividade tediosa e sucetível a erros. Por esta razão, VeriSC suporta a comparação automática dos resultados da verificação funcional. Esta comparação é responsabilidade do *Checker*.

- É baseado em transações. Os estímulos gerados pelo módulo *Source* possui um nível de abstração mais alto que o nível de abstração DUV, chamado de nível de transação. Uma transação é uma operação que se inicia num determinado momento no tempo e termina em outro, sendo caracterizada pelo conjunto de instruções e dados necessários para realizar a operação. O DUV operação no nível de sinais.

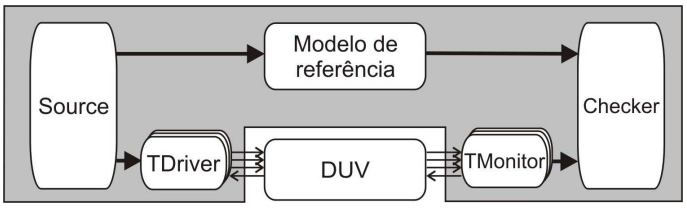

Figura A.5: Diagrama da verificação funcional da metodologia VeriSC.

VeriSC-RP

Os elementos que compõem o ambiente de verificação da metodolologia VeriSC-RP são os mesmos que compõem o ambiente de verificação de VeriSC. Além disso, VeriSC-RP também possui as quatros características mencionadas anteriormente. O formalismo HCPN e suas ferramentas associadas permitem que o ambiente de verificação seja dirigido por coberturas. A maneira como é isso é feito permite que diversos critérios de coberturas sejam especificados e analisados de uma maneira não intrusiva, isto é, sem a necessidade de inserir código nos elementos da Figura A.5. A principal ferramenta para HCPN, CPN Tools, possui a tecnologia de *monitores*. Um *monitor* CPN Tools, é um mecanismo utilizado para inspecionar e controlar uma rede HCPN. A grande vantagem é que este mecanismo não requer a inclusão de qualquer elemento no modelo do sistema, isto é, o comportamento do sistema e a monitoração operam de maneira independente, sendo possível até ligar e desligar a monitoração de acordo com as necessidades da simulação. Portanto, VeriSC-RP é dirigida por coberturas sendo que os critérios de cobertura são especificados como predicado do monitor *breakpoint*. Se esse o predicado for avaliado como verdadeiro, a monitoração vai parar a simulação.

Assim como VeriSC, VeriSC-RP não possui mecanismos automáticos para a geração de estímulos que atendem diretamente aos critérios de cobertura. De fato, esta abordagem automática de ambiente de verificação com auto-alimentação requer a aplicação de outras abordagens que extrapolam o escopo das linguagens de verificação de hardware. Fine e Ziv [37], por exemplo, fazem uso de algoritmos genéticos e redes *bayesianas* para que a geração de estímulos seja corvegente com os critérios de cobertura especificados. Portanto, o suporte a geração de estímulos de VeriSC-RP está restrita ao uso de funções aleatórias existentes nas ferramentas.

VeriSC-RP também possui a propriedade de auto-verificação. Esta auto-verificação é realizada de maneira trivial, por uma transição que recebe os dados advindos dos modelos e faz uma comparação dos mesmos. Com relação à operação em nível de transações, VeriSC permite a modelagem de dados através da linguagem de especificação CPN-ML, que é uma extensão da linguagem de programação funcional ML, com suporte a multi-conjuntos e operações sobre os mesmos.

A.4.2 A construção do ambiente de verificação

VeriSC

A metodologia VeriSC define uma implementação padrão para cada elemento do ambiente de verificação. Esta padronização é o princípio básico para a automação da construção do ambiente de verificação. Uma vez que os elementos são padronizados, é possível identificar quais características são *comuns* ao ambiente de verificação de qualquer IP e quais características precisam ser especificadas pelo engenheiro de verificação de acordo com a aplicação alvo do IP em desenvolvimento. As partes comuns são mantidas em moldes (*templates*). Estes moldes se tornam um ambiente de verificação de fato quando o engenheiro fornece os detalhes particulares do IP sendo verificado. A combinação do que é comum com é que é específico é realizado pela ferramenta eTBc[64].

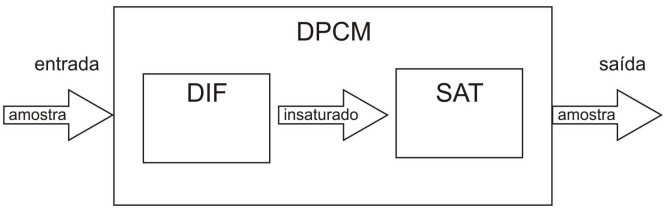

Figura A.6: Diagrama do DPCM decomposto em dois blocos: DIF e SAT.

O primeiro passo do engenheiro para construir o ambiente de verificação é definir os tipos de comunicação que irão compor o DUV. No caso do DPCM como proposto na Figura A.6, existem duas interfaces: *IN* e *OUT*. Para as duas interfaces, o tipo de dado é definido pela estrutura *dpcmSample*, isto é, o tipo de dado que entra no DPCM é o mesmo que sai. O tipo de dado que sai do sub-módulo DIF, no entanto, tem tipo diferente, *uns*, que é um valor insaturado. Este mesmo tipo *uns* serve como entrada do sub-módulo SAT. A partir destas informações, o arquivo que contém as partes específicas do DPCM pode ser descrito. O Código 14, nas linhas de 1 a 29, contém a especificação destes tipos (vide [64] para informações sobre a gramática da linguagem de especificação de ambientes de verificação da metodologia VeriSC).

Definidos os tipos de comunicação, o engenheiro precisa especificar a hierarquia de blocos do DUV, as interfaces que ele contém e as FIFOs que serão utilizadas para ligar os blocos do ambiente de verificação conforme apresentado no Código 15. A partir do arquivo formado

Algoritmo 14 : Descrição do tipo de comunicação da TLN

```
01 // Specifications for the DPCM converter.
02         // dpcm sample
03         struct <DPCM_DATA> {
04
05           trans {
06             signed [4] dpcmData;
07           }
08
09           signals{
10             signed [4] dpmcSgnl;
11             //signals for the handshake between comunicating blocks
12             bool valid;
13             bool inv ready;
14           }
15         }
16
17         // internal dpcm sample
18         struct <DPCM_UNSAT> {
19           trans {
20             signed [5] dpcmData;
21           }
22
23           signals{
24             signed [5] dpmcUnsaturedSgnl;
25             //signals for the handshake between comunicating blocks
26             bool valid;
27             bool inv ready;
28           }
29         }
```

pelos códigos 14 e 15, o engenheiro vai rodar a ferramenta eTBc, que vai gerar parcialmente o código do ambiente de verificação do DPCM. Para concluir o ambiente de verificação, o engenheiro vai precisar espeficar alguns detalhes, por exemplo, a função utilizada para a geração dos estímulos aleatórios. No caso do bloco DUV, a ferramenta vai gerar a sua casca a partir das informações sobre sinais e hierarquia extraídas dos códigos 14 e 15. A linguagem para a geração deste DUV parcial é Verilog.

Algoritmo 15 : Descrição da hierarquia e de interfaces do DPCM.

```
30
31          //module to perform the difference between two dpcm samples
32          module DIF{
33            input dpcmSample difIn;
34            output dpcmUnsaturated difOut;
35          }
36
37          //module to perform the saturation
38          module SAT{
39            input dpcmUnsaturated satIn;
40            output dpcmSample satOut;
41          }
42
43          //the DPCM module
44          module DPCM{
45            // main module interfaces
46            input dpcmSample sampleIn;
47            output dpcmSample sampleOut;
48
49            // unsaturated fifo between modules
50            fifo dpcmUnsaturated difToSat;
51
52            // link between modules via difToSat fifo
53            DIF difI(.difIn(sampleIn), .difOut(difToSat));
54            SAT satI(.satIn(difToSat), .satOut(sampleOut));
55          }
```

VeriSC-RP

O mesmo arquivo de especificação de eTBc pode ser usado para a construção do ambiente de verificação em VeriSC-RP. Neste caso, cada parte que compõe o arquivo será usada para gerar a rede de Petri de cada elemento do ambiente de verificação. A idéia dessa geração é simples e segue o mesmo princípio de VeriSC. Em VeriSC, este arquivo de especificação

é compilado juntamente com os moldes em SystemC e o resultado da compilação é o ambiente de verificação também em SystemC. Em VeriSC-RP este arquivo de especificação é compilado juntamente com moldes em HCPN e o resultado da compilação é o ambiente de verificação em HCPN.

O processo de compilação consiste em converter os blocos de comando *struct* para *colour sets* da rede. Seguindo o mesmo princípio, cada *module* é convertido em uma transição de substituição de HCPN, sendo que cada *input* é uma porta de entrada e cada *output* é uma porta de saída. Uma declaração do tipo FIFO corresponde a um *socket* de entrada e saída entre os módulos comunicantes. No caso do DPCM, nas linhas 53 e 54 do código 15, ocorre a associação de lugares *sockets* a lugares portas. Assim, o lugar *difIn* da sub-página DIF vai se associar com o *socket sample*, da super-página DPCM. O lugar *difOut* vai se associar com o *socket difToSat*. Este *difToSat* será o *socket* com a porta de entrada *satIn* da sub-pagina SAT. Por último, a porta de saída de SAT, *satOut* terá como *socket* correspondente o lugar *sampleOut*. A Figura A.7 contém o ambiente de verificação gerado para VeriSC-RP. O conteúdo de cada sub-pagina será discutido nas subseções seguintes.

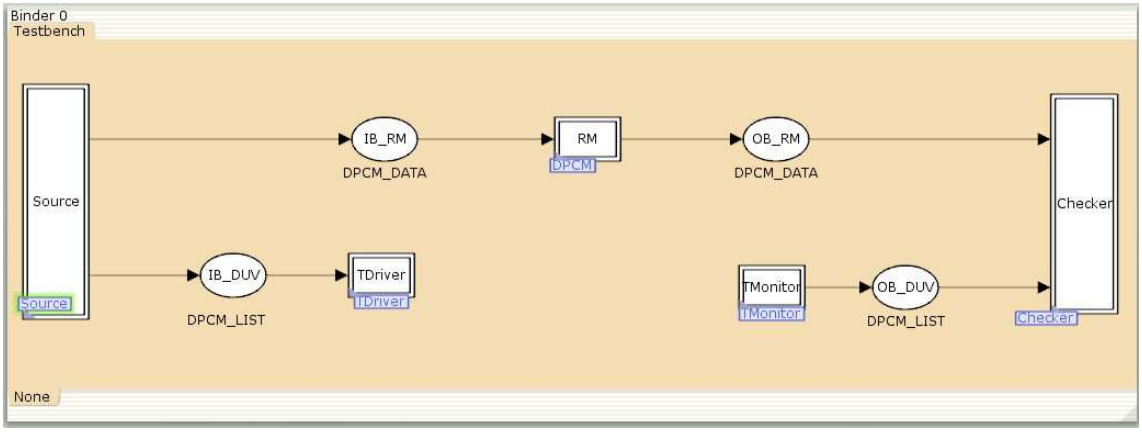

Figura A.7: Rede do ambiente de verificação.

A.4.3 O bloco *Source*

A.4.4 VeriSC

O bloco *Source* é responsável por gerar os estímulos da verificação funcional. Estes estímulos são gerados com abstração de transação. Quatro tipos de estímulos são recomendados

por VeriSC: estímulos direcionados, estímulos de situações críticas, estímulos aleatórios e estímulos reais.

Os estímulos direcionados são escolhidos manualmente pelo engenheiro de verificação. Eles são interessantes em estágios iniciais da verificação funcional para se constatar a presença de erros crassos de *design*. Estímulos de situações críticas são aqueles responsáveis por estimular funcionalidades nas quais existe alguma suspeita de existência erro. À medida que o número de funcionalidades do IP aumenta, torna-se impraticável para o engenheiro fazer a previsão dos estímulos que são capazes de exercitar as diversas partes do *design* do IP. Por esta razão, o bloco *Source* deve ser capaz de gerar estímulos aleatórios que exercitem pontos do design que não foram antecipados pelo engenheiro de verificação. Por último, os estímulos reais são selecionados de uma situação real da aplicação do circuito digital em desenvolvimento.

A ferramenta eTBc gera a grande maioria do código do *Source*. Cabe ao engenheiro especificar a função para geração de valores aleatórios e os arquivos que contêm os estímulos direcionados, de situações críticas e reais.

VeriSC-RP

A geração automática do bloco *Source* em VeriSC-RP possui as mesmas características de VeriSC. A Figura A.8 é um exemplo do bloco gerado para o modelo do DPCM. Cada transição da rede representa a geração de um tipo de estímulo. Na transição *Random* ocorre a geração dos estímulos aleatórios. A geração desses estímulos é realizada por um código associado a esta transição. O engenheiro pode especificar as diversas funções de distribuição existentes para *CPN Tools*, tais como *Poisson*, *Bernoulli* etc.

A transição *Directed* é responsável pela geração dos estímulos direcionados. Neste caso, o engenheiro determina quais são os estímulos direcionados através da marcação do lugar DS. Uma vez que os estímulos aleatórios tenham sido gerados, os estímulos direcionados são agregados à verificação funcional mediante o disparo da transição *Directed*. Seguindo este mesmo princípio, os estímulos de situações críticas e reais são gerados. Caso o engenheiro não queira especificar algum destes estímulos, basta que ele declare a marcação inicial do lugar como sendo uma lista vazia, conforme o caso do lugar *RDS*. Quando a transição *RealData* dispara, todo o conjunto de estímulos é transmitido para o bloco *TDriver* através do

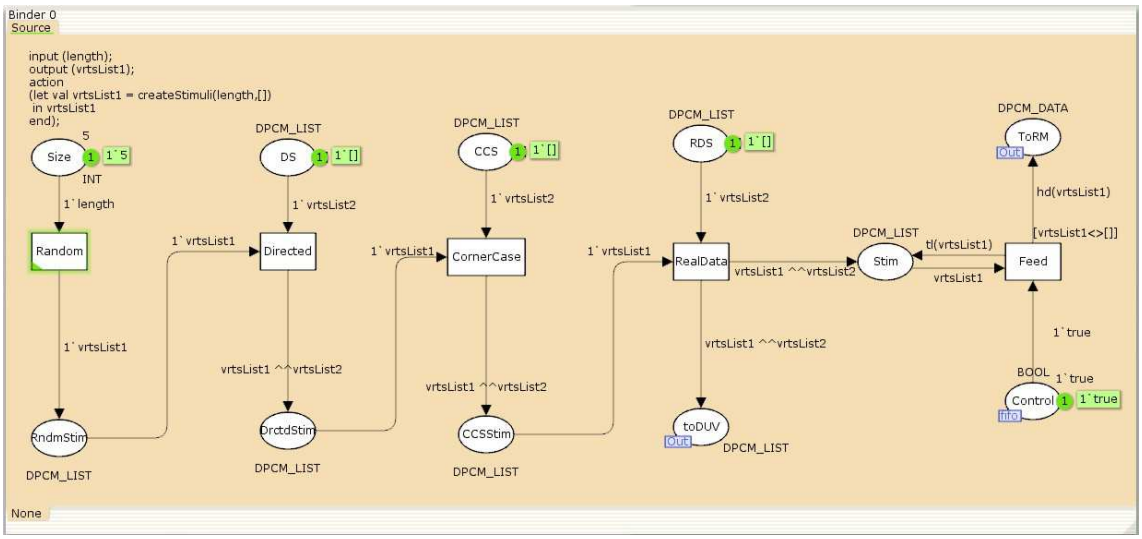

Figura A.8: Rede do bloco *Source*.

lugar *toDuv*. A transição *Feed* e o lugar *control* existem para que os dados do *Source* sejam fornecidos para o Modelo de Referência respeitando-se a estrutura do tipo *FIFO*.

A.4.5 Os blocos Modelo de Referência e DUV

VeriSC

O Modelo de Referência é a implementação ideal do sistema. Assim, ao receber estímulos, ele deve produzir respostas corretas de acordo com a especificação do sistemas. Naturalmente, a especificação deste bloco vai mudar de acordo com o propósito do circuito digital a ser desenvolvido. Logo, não existe como prever um modelo de referência para cada sistema a ser desenvolvido. A ferramenta eTBc, no entanto, gera um molde que contém a "casca" do modelo de referência. Esta "casca" corresponde ao código que trata da comunição com os blocos vizinhos, tais como entrada na FIFO do dados que chegam do *Source* e saída dos dados na FIFO que são passados para o bloco *Checker*. A partir deste molde, o engenheiro vai concluir o modelo de referência de acordo com a especificação do sistema.

VeriSC suporta modelo de referência em qualquer linguagem que seja capaz de se comunicar com código *SystemC* no nível de transações, pois os demais elementos do ambiente de verifição também estão codificados nesta linguagens. Esta comunicação pode ocorrer através de FIFOs ou usando linguagem de programação que permita o acesso os recursos de *pipe*

do sistema operacional. Atualmente, o molde do modelo de referência é gerado também em SystemC.

O DUV não faz parte do ambiente de verificação. O DUV é o *design* sendo verificado. Como ele se comunica com a ambiente de verificação, é importante que sua implementação esteja de acordo o ambiente de ambiente de verificação desde o seu início. A implementação do DUV que não considera o ambiente de verificação e vice-versa podem levar a retrabalho para adequação de uma das partes. Da mesma maneira que o modelo de referência, não é possível prever o conteúdo do DUV. A aplicação alvo do sistema é quem vai orientar a composição desse módulo. Conseqüentemente, a automação da geração desse bloco está restrita a geração de uma "casca" correspondente a parte de comunicação com o ambiente de verificação. Como este bloco opera no nível de sinais e o modelo de referência opera no nível de sinais, a parte de comunicação do DUV corresponde a geração de elementos de comunicação com os blocos *TDriver* e *TMonitor*. A linguagem do código gerado é Verilog.

VeriSC-RP

VeriSC-RP também suporta a geração do molde do modelo de referência em HCPN, considerando inclusive a decomposição hierárquica deste bloco. A "casca" deste bloco vai conter a declaração das transições de substituição, dos *sockets* e das portas. A Figura A.9 é molde do modelo de referência do DPCM gerado pelo protótipo para VeriSC-RP. Os moldes de detalhamentos das subpáginas DIF e SAT estão ilustrados nas figuras A.10 e A.11 respectivamente.

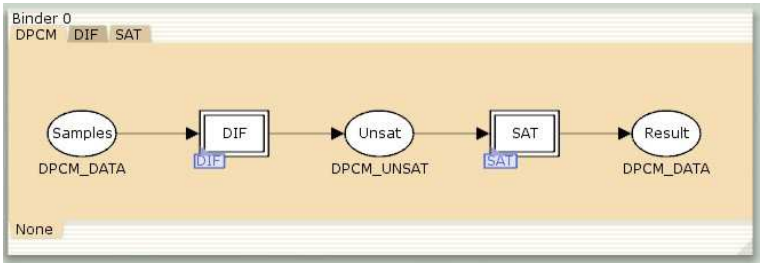

Figura A.9: Molde do modelo de referência do bloco DPCM.

A parte de geração do molde do DUV permanece tal como é em VeriSC. Isto porque a linguagem deste molde é Verilog e não HCPN. O protótipo de geração de ambientes de verificação ainda não trata da geração deste molde.

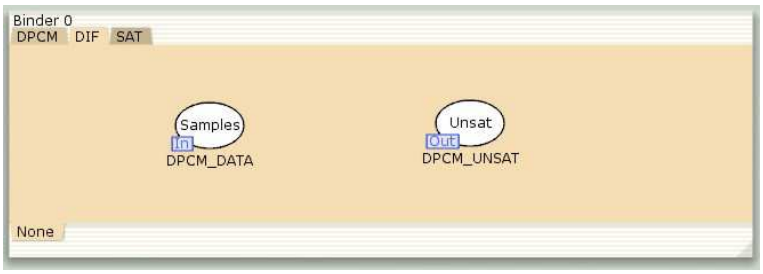

Figura A.10: Molde do modelo de referência da subpágina DIF.

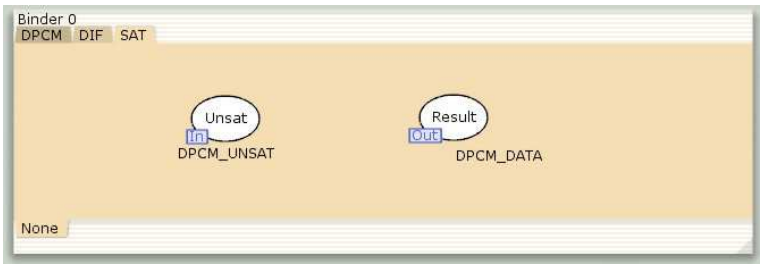

Figura A.11: Molde do modelo de referência da subpágina SAT.

A.4.6 Os blocos *TDriver* e *TMonitor*

VeriSC

Um bloco *TDriver* é responsável pela comunicação do *Source* com uma determinada interface de entrada do DUV. Uma interface de entrada é o canal pelo qual o DUV recebe dados de outros blocos. Assim, um ambiente de verificação pode possuir um ou mais blocos *TDriver*, dependendo da quantidade de interfaces de entrada do DUV. Cada interface de entrada possui seu protocolo de comunicação entre os blocos envolvidos. Este protocolo precisa ser parcialmente implementado pelo engenheiro de verificação.

O papel complementar ao do *TDriver* é executado pelo bloco *TMonitor*. Assim, *TMonitor* tem o papel de converter estímulos na forma de sinais para transações . Igualmente, para cada interface de saída do DUV, vai existir um bloco *TMonitor*. Em resumo, para o molde do *TMonitor*, o engenheiro de verificação precisa especificar as mesmas informações que são especificadas para o *TDriver*.

VeriSC-RP

Em VeriSC-RP, também existe um *TDriver* e um *TMonitor* para cada interface de comunicação do bloco DUV. De fato, *TDriver* e *TMonitor* têm o mesmo papel que possuem em VeriSC. O que muda é o protocolo de comunicação. Em VeriSC, as comunicações do *Source* com DUV e do *Source* com o Modelo de Referência são realizadas estímulo por estímulo, pois a verificação funcional como um todo ocorre no mesmo espaço de memória. Em VeriSC-RP temos que esta comunicação vai ocorrer em lotes. Isto porque o ambiente de verificação funcional, em HCPN, não pode ser simulado no mesmo espaço de memória que o DUV, em Verilog ou SystemC. Assim, o bloco *TDriver* recebe todos os estímulos do *Source* para depois fazer a conversão deste estímulos para a abstração de sinais. Os sinais obtidos são gravados em um arquivo de entrada do simulador do DUV. Seguindo o mesmo princípio, o simulador Verilog vai gravar todos os resultados da simulação do DUV em outro arquivo, que funciona com entrada do bloco *TMonitor*. O *TMonitor* vai ler estes dados e convertê-los para transações, passando-as para o *Checker*.

O bloco *TDriver* está ilustrado na Figura A.12. A conversão dos dados do *Source* e gravação dos dados em arquivo ocorre no disparo da transição *TDriver*. O código da conversão e da gravação está registrado como um monitor *CPNTools* desta transição. O código para conversão de valores é produzido a partir das informações fornecidas no arquivo de especificação do ambiente de verificação, tal como no trecho da linha 3 à linha 15 do Código 14. O bloco *TMonitor* está ilustrado na Figura A.13. A recepção dos dados vindos do DUV aparece como código da transição porque *CPN Tools* não possui monitor específico para leitura de arquivos.

A.4.7 O bloco *Checker*

VeriSC

O bloco *Checker* é responsável por comparar automaticamente os resultados advindos do Modelo de Referência e do DUV. Se valores diferentes são detectados na comparação, o engenheiro é informado desse erro mostrando-se para ele o valor esperado e o valor calculado pelo DUV. Esta informação é útil para auxiliar o engenheiro no processo de depuração. A geração automática deste bloco exige intervenção do engenheiro, isto é, ela é realizada por

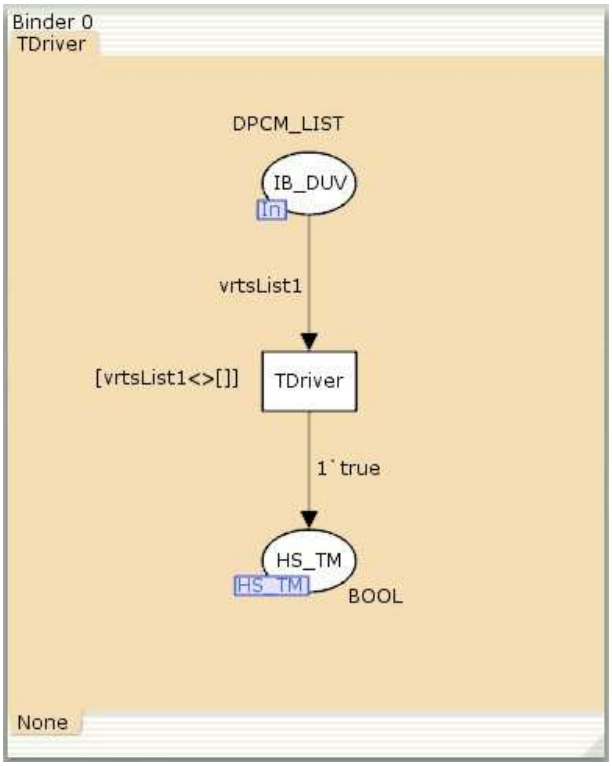

Figura A.12: Rede do bloco *TDriver*.

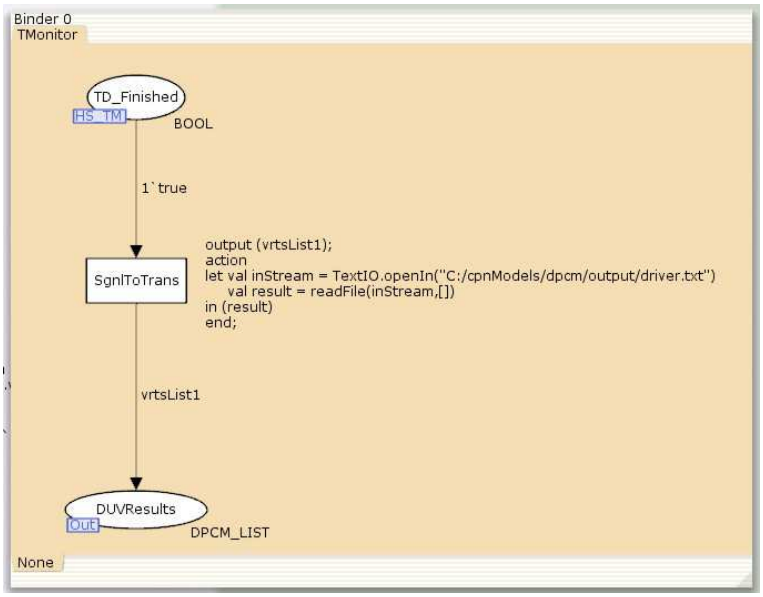

Figura A.13: Rede do bloco *TMonitor*.

completo pela ferramenta de geração de ambiente de verificação eTBc.

VeriSC-RP

Em VeriSC-RP, o bloco *Checker* possui as mesmas funcionalidades. A sua geração também é realizada de maneira automática, sem a intervenção do engenheiro de verificação. A rede de Petri deste bloco está ilustrado na Figura A.14. Os resultados gerados pelo Modelo de Referência ficam guardados no lugar *fromRM* e os resultados gerados pelo DUV ficam guardados no lugar *fromDUV*. A comparação nos casos em que os resultados são iguais é modelada pela transição *areEquals*, sendo que o par comparado é armazenado no lugar *success*. Quando os valores são diferentes, a comparação é realizada pela transição *areDifferents*, sendo que o par comparado vai ficar armazenado no lugar *errors*. Este par de valores discrepantes também pode ser gravado em arquivo através de um monitor *CPN Tools* que existe na transição *areDifferents*.

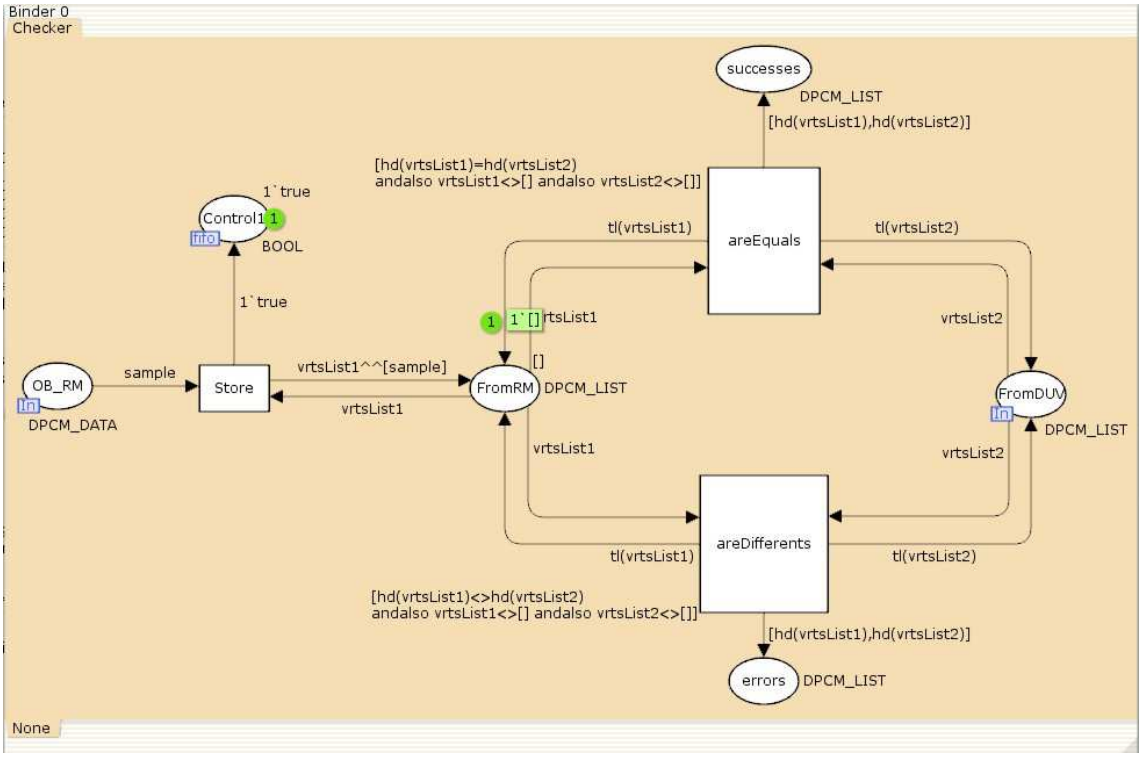

Figura A.14: Rede do bloco *Checker*.

A.4.8 O fluxo da metodologia

O fluxo de atividades da metodologia VeriSC trata da implementação do ambiente de verificação antes da implementação do DUV. Esta abordagem possui várias vantagens em relação a abordagem tradicional, que espera a conclusão da implementação do DUV para depois implementar o ambiente de verificação. Detalhes sobre estas vantagens podem ser encontrados no trabalhos [32; 31]. A seguir são citadas de maneira resumida algumas dessas vantagens.

1. O custo de integração do ambiente de verificação com o DUV é reduzido consideravelmente, podendo até ser nulo, pois o ambiente de verificação é gerado antes do DUV e a partir de uma especificação única para ambos. Esta especificação inicial única promove uma integração de interfaces entre as partes comunicantes do processo de verificação funcional.

2. A verificação funcional pode ser usada desde o início do projeto do DUV, economizando tempo de projeto. Sem o ambiente de verificação, o projetista do DUV vai ter que realizar testes durante o desenvolvimento do DUV ou então densenvolver todo o DUV para depois fazer a verificação funcional.

3. O ambiente de verificação pode passar por uma fase de depuração sistemática e bem elaborada de modo a diminuir significativamente a quantidade de erros nos elementos do ambiente de verificação.

4. Os elementos do ambiente de verificação podem ser reusados durante o seu desenvolvimento. Circuitos digitais são complexos e exigem mecanismos de decomposição de design. VeriSC define um fluxo que permite a decomposição dos *designs* de modo que os elementos desenvolvidos para a verificação funcional de cada parte de design seja reutilizada na composição ambiente de verificação funcional do *design* completo.

O uso de redes de Petri coloridas para composição do ambiente de verificação deve, portanto, preservar estas vantagens deste fluxo de atividades. No decorrer dessa seção, será apresentado esse fluxo original e será discutida a sua adaptação para o caso em que o ambiente de verificação funcional é descrito usando-se redes de Petri coloridas hierárquicas. O fluxo de VeriSC é constituído por quatro passos básicos e cada um deles será discutido nas

subseções seguintes. Deve ficar claro para o leitor que a proposta original de VeriSC assume a existência do Modelo de Referência antes do início da implementação ambiente de verificação. Não há textos que tratem do uso de VeriSC em projeto sem o modelo de referência desde o início da verificação funcional.

VeriSC

Primeiro passo: construção do ambiente de verificação completo A primeira atividade a ser realizada pelo engenheiro de verificação é a geração do ambiente de verificação para o DUV completo, isto é, sem decomposição. Esta atividade é constituída por outras três subatividades:

1. Teste de comunicação do Modelo de referência com o ambiente de verificação. Para estimular o modelo de referência, um bloco auxiliar, chamado de *pré-source*, é construído. Este bloco auxiliar já considera a geração de estímulos direcionados e estímulos aleatórios. Para receber os resultados produzidos pelo modelo de referência, outro bloco auxiliar, chamado de *Sink*, é construído.

2. Construção dos blocos *Source* e *Checker*. Nesta passo, estes blocos também são validados. Para isso, o bloco Modelo de Referência é replicado, sendo que a réplica faz o papel do DUV, que ainda não existe. Assim, o *Source* estimula os dois modelos de referência e o *Checker* verifica se os resultados produzidos são iguais. Adicionalmente, pode-se injetar erros na réplica do modelo de referência para analisar a habilidade do *Checker* de se detectar tais erros.

3. Construção e teste dos blocos *TDriver* e *TMonitor*. Para realizar o teste sobre estes elementos sem a implementação do DUV, a réplica do modelo de referência continua sendo usada, porém ela vai receber estímulos de uma réplica do bloco *TMonitor*. Este *TMonitor* vai receber estímulos do *TDriver* original, convertendo-os da forma de sinais para a forma de transações. Na saída da réplica do modelo de referência, será colocada uma réplica do bloco *TDriver* para viabilizar a comunicação com o bloco *TMonitor* original.

Segundo passo: decomposição hierárquica do Modelo de Referência Neste passo o engenheiro vai decompor o Modelo de Referência seguindo a mesma estrutura hierárquica do DUV. Cada bloco resultante da decomposição hierárquica passa a ser tratado como um bloco independente, sendo que para cada bloco será constituído um ambiente de verificação. Porém, antes disso, a decomposição hierárquica é testada para se certificar de que nenhum erro foi inserido nessa fase. Este teste é feito fornecendo-se estímulos para os modelo de referência sem decomposição e com decomposição sendo a comparação dos resultados realizada pelo bloco *Checker*.

Terceiro passo: ambiente de verificação para bloco do DUV O terceiro passo trata de construir o ambiente de verificação para cada bloco da decomposição hierárquica do DUV. Para cada ambiente de verificação criado, devem ser seguidos as mesmas regras e os mesmos passo para contrução do ambiente de verificação completo. Neste passo, os blocos *TDriver* e *TMonitor* do ambiente de verificação completo podem ser reusados, pois tais interfaces são mantidas na decomposição. As interfaces de comunicação entre blocos da decomposição hierárquica também podem ser reusadas para a construação do ambiente de verificação individual de cada bloco. Seguindo este princípio, a metodologia pode ser usada para qualquer número de blocos e qualquer topologia de decomposição hierárquica.

Quarto passo: substituição do DUV completo No quarto é último passo, todos os blocos que compõem o DUV são ligados em um único bloco. Em seguida, deve-se testar se erros não foram inseridos nesta ligação. O bloco completo resultante da ligação de cada sub-bloco do DUV é usado no ambiente de verificação criado no primeiro passo.

VeriSC-RP

Até o momento da escrita deste artigo, não encontramos nenhum trabalho sobre verificação funcional de circuitos digitais que faça uso de redes de Petri para a descrição do ambiente de verificação. Por este motivo, na parte prática da adaptação, não foi possível considerar a existência prévia de um modelo de referência em redes de Petri coloridas. Conceitualmente, assumindo-se a existência de um modelo de referência em redes de Petri coloridas, é possível seguir o mesmo fluxo de VeriSC. Porém, por ausência de experimentação, não será possível

discutir com detalhes, por exemplo, o custo de se adaptar o modelo de referência para a mesma hierárquia do DUV.

Em resumo, os passos originais de VeriSC visam a construção incremental do ambiente de verificação em um processo que permite a validação dos elementos do ambiente de verificação. Esta construção hierárquica é garantida em VeriSC-RP através do formalismo de redes de Petri coloridas hierárquicas e suporte ferramental associado. Com relação à validação dos elementos do ambiente de verificação, além do que VeriSC original oferece, em VeriSC-RP o engenheiro terá a sua disposição técnicas formais, tais como análise de espaço de estados, para se certificar de que o ambiente de verificação não possui erros. Além disso, durante experimentos de verificação com o exemplo do DPCM foi possível automatizar alguns dos passos discutidos anteriormente. Por exemplo, foi possível automatizar a replicação do modelo de referência no primeiro passo assim como a associação desta réplica com as réplicas do *TDriver* e do *TMonitor*.

A.4.9 Análise de cobertura

Por se tratar de uma técnica baseada em simulação, a qualidade da verificação funcional depende da habilidade do engenheiro de verificação de detectar se todas as funcionalidades do IP foram exercitadas. O mecanismo empregado pelos engenheiros para se fazer essa detecção se chama cobertura funcional. Na cobertura funcional, o engenheiro especifica um conjunto de metas que devem ser atingidas durante a verificação funcional. Uma vez que estas metas são atingidas, o processo de verificação funcional pode ser encerrado e uma nova etapa na construção do hardware pode ser iniciada.

A metodologia VeriSC provê análise de cobertura através de uma biblioteca C++, chamada de *BVE-COVER (Brazil-IP Verification Extension*, que pode ser usada com as demais bibliotecas SystemC. Como VeriSC é uma metodologia *black-box*, isto é, a verificação funcional não trata de detalhes internos do bloco DUV, a análise de cobertura é realizada observando-se as interfaces disponíveis e também o modelo de referência.

O primeiro passo para se implementar a cobertura funcional é definir um modelo de cobertura. Este modelo é um documento que contém a especificação da implementação da cobertura funcional. Ao se construir um modelo de cobertura, o engenheiro deve estar ciente dos objetivos esperados durante a verificação funcional. Em geral, estes objetivos são rela-

ções de resultados que o dispositivo deve produzir, relação de resultados que o dispositivo não deve produzir e situações que devem ser ignoradas na análise de cobertura Um possível modelo de cobertura para o DPMC hierárquico está descrito na Tabela A.1. Esta tabela registra os valores que devem ser alcançados durante a verificação funcional. Em particular, as duas variáveis principais do problema estão sendo observadas: a amostra de entrada e a amostra de saída. Limite inferior é valor mínimo que a diferença entre duas amostras consecutivas pode possuir e limite superior é valor máximo que a diferença entre duas amostras consecutivas pode possuir Embora não esteja registrado na Tabela A.1, o engenheiro poder definir critérios de cobertura em relação aos valores que são passados entre os blocos comunicantes.

Atributo	Amostra de Entrada	Amostra de Saída
Valores esperados	Limite Inferior - 1; Limite Inferior; Limite Inferior + 1; Limite Superior - 1; Limite Superior; Superior + 1	Limite Inferior; Limite Inferior + 1; Limite Superior; Limite Superior - 1

Tabela A.1: Modelo de cobertura do DPCM.

A biblioteca *BVE-COVER* é composta por quatro módulos diferentes: *BVE-COVER Bucket*, *BVE-COVER Illegal*, *BVE-COVER Ignore* e *BVE-COVER Cross-coverage*.

O módulo *BVE-COVER Bucket* é responsável por analisar as funcionalidades que devem ser cobertas na verificação funcional. As funcionalidades são especificadas como *baldes* que devem ser enchidos durante a simulação. Para evitar que a simulação fique eternamente como conseqüência de uma funcionalidade que de fato não existe no *design*, o tempo de simulação também é usado como critério de parada para este módulo. As funcionalidades, assim como nos demais módulos de cobertura, são especificadas por instrumentação de código, tal como no exemplo do Código 16. Neste exemplo, está especificado que 10 amostras de saída do DPCM devem ser iguais ao limite superior de saturação e que outras 10 devem ser iguais ao limite inferior de saturação.

O módulo *BVE-COVER Ignore* é responsável por analisar os *buracos* de cobertura. Um buraco de cobertura é um conjunto de funcionalidades que ficou descoberta durante a verificação funcional. Há dois tipos de buracos de cobertura: válido e inválido. Um buraco de

Algoritmo 16 : Instrumentação de código para análise de cobertura.

```
01          BVE_COVER_BUCKET Cv_bucket_Limit;
02
03          Cv_Bucket_Limit.begin();
04            BVE_COVER_BUCKET(Cv_bucket_Limit, (sampleOut == UPPER_LIMIT,10));
05            BVE_COVER_BUCKET(Cv_bucket_Limit, (sampleOut == LOWER_LIMIT,10))
06          Cv_Bucket_Limit.end();
```

cobertura válido é um requisito que faz parte do modelo de cobertura mas que não foi observado na verificação funcional. Um buraco de cobertura inválido é um requisito funcional que não deve fazer parte do modelo de cobertura. O módulo *BVE-COVER Ignore* é usado para evitar os buracos de cobertura inválidos, tornando mais preciso a análise de buracos de cobertura válidos.

O módulo *BVE-COVER Illegal* analisa a execução de funcionalidades que não estão de acordo com a especificação do sistema. Este módulo funciona como asserções que são inseridas no código e caso sejam executadas, elas são reportadas ao engenheiro de verificação.

A cobertura *Cross-coverage* é um modelo definido pela permutação de funcionalidades que devem ser exercitadas. A idéia é avaliar se critérios anteriormente avaliados separadamente são satisfeitos em conjuntamente com os demais. Os baldes de *BVE-COVER Bucket* são usados para se construir matrizes multidimensionais. A cada novo evento, o simulador verifica se algumas dessas informações cruzadas ocorreram.

VeriSC-RP

A especificação de critérios de cobertura em VeriSC-RP se dá através de monitores *CPN Tools*. Estes monitores permitem ao engenheiro extrair informações sobre os elementos do ambiente de verificação e até mesmo parar uma simulação. Na Figura A.15 está especificado um critério de cobertura do tipo *Bucket*. O módulo *Bucket* é responsável por parar a verificação funcional caso os baldes estejam cheios. Para que critérios de cobertura deste módulo tenham o mesmo efeito em VeriSC-RP, eles são especificados como monitores do tipo *breakpoint*. Assim, quando a função *pred* do monitor for satisfeita, a verificação funcional vai parar. O balde da Figura A.15 especifica que 2 amostras de valor 2, produzidas pelo DUV, devem ser enviadas ao bloco *Checker*. Os demais módulos de cobertura também são modelados através de monitores, especialmente os monitores de coleta de dados.

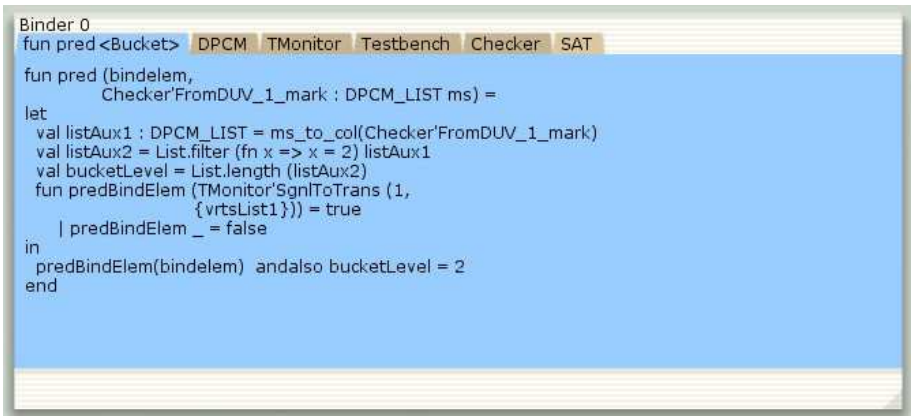

Figura A.15: Código para análise de cobertura do tipo *bucket*.

A.4.10 Considerações Finais

Esta Seção foi dedicada à discussão do uso de redes de Petri coloridas hierárquicas e ferramentas associadas na metodologia de verificação funcional VeriSC. O objetivo desta discussão foi esclarecer como o uso do formalismo de redes de Petri influencia na metodologia. Em resumo, as principais características de VeriSC original são preservadas. O que muda de fato, é *forma* de como cada atividade é realizada, pois VeriSC original é baseada em C-C++ e SystemC. O que não foi preservado da metodologia original foi o protocolo de comunicação entre o trio *TDriver*, *DUV*,*TDriver*, pois o simulador do DUV não pode operar no mesmo espaço de memória do ambiente de verificação. Assim, em VeriSC-RP, por questão de eficiência, a comunicação entre o trio *TDriver* , *DUV*,*TDriver* deve ocorrer em lotes, em vez acontecer estímulo por estímulo.

A parte de análise de cobertura em VeriSC-RP tem uma característica que VeriSC não tem. Em VeriSC-RP a análise de cobertura não é instrusiva como é em VeriSC. Os critérios de cobertura são especificados e analisados de forma independente da especificação do modelo e do seu engenho de simulação. Isto é uma vantagem, pois promove a separação das especificidades do modelo das especificidades da cobertura, permitindo até que o análise de cobertura seja ligada e desligada conforme a necessidade do engenheiro de verificação.

A.5 Considerações Finais

Neste trabalho, apresentamos uma extensão da metodologia de verificação funcional VeriSC. Esta extensão foi avaliada em uma modelagem dos blocos PIACDC e QI do decodificador de vídeo MPEG 4 [69], originalmente desenvolvido usando-se VeriSC e o modelo de referência em linguagem C, dado pelo XVID [2]. Estes blocos foram escolhido em função da dificuldade reportada pelos engenheiros de decompor tal modelo de referência com a mesma granularidade esperada para o design em linguagem de descrição de hardware. Está experiência está reportada no artigo [70].

Apêndice B

Cross Coverage Analysis using Association Rules Mining

B.1 Introduction

Functional verification is a technique to demonstrate that the intent of a hardware design is preserved in its implementation [18; 88; 65]. In fact, there is no consensual functional verification methodology in the digital circuit industries. Each industry tailors its methodology according to the type of digital circuit to be produced, the resources that are available and constraints that are imposed by the project. However, as a rule, every functional verification methodology comprises four basic components: i) a set of specifications that design must comply with; ii) the Register Transfer Level (RTL) design under verification (DUV); iii) a simulation mechanism to judge the DUV against its set of specifications; iv) a mechanism to estimate the level of confidence achieved during the functional verification process. Except by the DUV, the remaining components are enclosed in an environment called *testbench*.

The fourth component is based on a *coverage model*, which work as list of goals that must be achieved during the simulation. Cross Product Coverage (CPC), for instance, is a kind of functional coverage in which the engineer analyses the occurrence of combinations of values during the functional verification. An example of goal that can be analyzed with CPC is if all commands of a calculator were executed in all ports. Traditionally, the engineers specify CPC by using matrices, which report the information if the desired combinations occurred and how many times they were simulated. CPC can be impracticable if more than two items

are related because the number of possible value sets grows factorially with the number of crossed items [18]. In this work, we present an alternative for the use of matrices to perform CPC analysis during the functional verification of hardware designs. We are performing CPC analysis by exploiting the techniques of association rules mining, originally developed for discovering interesting relations between variables in large databases [80]. An association rule is a pattern $X \rightarrow Y$ such that X and Y are sets of values. A typical application of this kind of mining is the cross-selling in on-line book stores, which suggest for the user that *"Customers who bought the book X also bought the book Y"*. By using association rules mining, we are getting a more flexible and practical tool to deal with CPC analysis that relates more than two elements. Furthermore, we can analyze the *frequency* in which certain combinations occur during the simulation. Such tool can be used to investigate and improve the quality of stimuli generation and coverage specification. We exploit mined association rules to enhance the stimuli generation and coverage specification during the hierarchical composition of design blocks. We check if a design block was sufficiently simulated with regard its neighbor blocks. By means of experimental results, we show that our approach can improve the scores given by others structural coverage models, such as code coverage and branch coverage.

In [38], Fountain et al reported an application of Data Mining and Decision Analysis to the problem of Die-Level functional test (DLFT) in integrated circuit manufacturing. It describes a decision-theoretic approach to DLFT, in which historical test data is mined to create a probabilistic model of die failure patterns. The Data Mining is used to analyze data to build and update probabilistic models and the Decision-theoretic Control is used to make decisions based on those models. Braun *et al* [25] employ data mining techniques to deal with the problem of reach corner-cases during the functional verification. This work is a kind of automated Coverage Directed Generation [37]. The main purpose of this approach is to hit uncovered areas of the design without human interaction. The feedback given by coverage measurements collected during the simulation is used to adjust the parameters and biases of stimuli generation.

Our approach was implemented for the VeriSC methodology [32; 15]. VeriSC is a SystemC-based [7] methodology that allows the generation of the complete running test-bench before the implementation of the DUV has been started. Hence, the design can be

verified in all necessary phases of its implementation, mainly at the beginning of the DUV implementation. Furthermore, the VeriSC methodology can reuse its own elements to implement the testbenches, to perform a self-test and to assure that the testbenches contain no errors.

The remaining of this paper is structured as follow. Section B.2 presents the main concepts about VeriSC. Sections B.3 and B.4 contain the explanation of our strategy to mine association rules during the functional coverage specification. Section B.5 presents case studies that quantify the contribution of our work. Finally, Section B.6 contains the final remarking.

B.2 VeriSC Methodology: an Overview

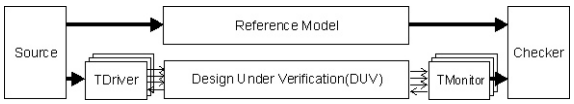

Figura B.1: Testbench to functional verification

The general testbench scheme used in VeriSC is shown in Figure B.1. It is composed of the following blocks: Source, TDriver(s), TMonitor(s), Reference Model (RM) and Checker. RM gives the specifications that design must comply with. The testbench functionality is executed inputting data into the DUV and the RM, capturing the outputs and comparing automatically if both outputs are equivalent. The synchronization mechanism of the testbench is implemented by means of First-In-First-Out queues (FIFOs). This testbench structure is intended to be used only for synchronous designs.

The Source is responsible for providing transaction level (TL) data to the DUV and to the RM. The Source is connected to the RM and to the TDriver by means of FIFOs. Each FIFO is responsible to maintain the order of the data to be compared. There is one FIFO for each interface. The same numbers of FIFOs are going to the RM and to the TDrivers. These stimuli can be random or directed, depending on the specific verification requirements.

The TDriver receives TL data from the Source, transforms it in specified protocol signals and passes them along with the required data to the DUV. Each output interface from the DUV has one TMonitor. The TMonitor is the bridge between the signal data and transac-

tions. Thus, it is responsible for receiving the protocol signals from the DUV and transforming them into TL data. The TMonitor puts the TL data into a FIFO and passes them to the Checker. The Checker is responsible for comparing TL data coming from the RM with TL data coming from TMonitor(s) to see if they are equivalent. The checker will automatically compare the outputs from RM and DUV and prints error messages if they are not equivalent.

The coverage-driven simulation in VeriSC is implemented by the BVE-COVER library. The coverage model has the semantic definition of a list of attributes and a set of values for these attributes. The verification engineer defines the attributes and values according to the verification plan.

All the important functionalities, which should be covered in a simulation, must be specified using the Coverage-bucket. The functionalities are specified using code instrumentation as shown in Code 17. The verification engineer should specify which functionality should be executed and how many times it should be executed to reach 100% of this functionality. In order to show how the library works, in the following example, the simulation will stop when the addresses 0xff and 0x80 are reached 5 times each. It is an example case, but it could be used anyway to reach other results. Each time a functional coverage point is executed, it brings up to date the coverage percentage. The Bucket calculates the percentage of the desired total coverage that has been covered. The algorithm to calculate the percentage uses the formula:

$$coverage = \frac{\sum_{i=1}^{N} coverage_i}{N}$$

where $coverage_i$ is a particular coverage measurement of each coverage point and N is the number of coverage points. The Coverage-bucket is also responsible for measuring the progress of the simulation. When all coverage points have been 100% covered the simulation stops.

Algoritmo 17 : BVE-COVER Bucket

```
BVE_COVER_BUCKET Cv_bucket_addr;

Cv_bucket_addr.begin( );
  BVE_COVER_BUCKET(Cv_bucket_addr, (addr==0xff, 5));
  BVE_COVER_BUCKET(Cv_bucket_addr, (addr==0x80, 5));
Cv_bucket_addr.end( );
```

The Cross-coverage is a model whose space is defined by the full permutation of all values of all attributes, more precisely known as multi-dimensional matrix coverage. Cross-coverage is based on the cross information from defined buckets. One example is a Cross-coverage bucket that contains commands and address. The Cross-coverage can verify if all commands occurred in all addresses.

Some or all buckets could be selected to create the Cross-coverage. With this information, a table is created with the crossing of the buckets information. At each new event, the simulator verifies if some of this crossing information occurred. In case of an occurrence, these data are uploaded in the table. The code instrumentation is done as shown in Code 18.

Algoritmo 18 : BVE-COVER Cross-coverage

```
BVE_COVER_CROSS_COVERAGE  Cv_cc;

Cv_cc.begin( );
  BVE_COVER_CROSS_COVERAGE(Cv_cc, Cv_bucket_addr);
  BVE_COVER_CROSS_COVERAGE(Cv_cc, Cv_bucket_cmd);
Cv_cc.end( );
```

B.3 Mining Association Rules

Figure B.2 illustrates the phases of mining association rules during the functional verification. It starts by collecting information about the simulation. At this point, we employ the SystemC Verification (SVC) Library to capture transaction-level information that is used to create functional coverage metrics, such as timing information and attribute information. VeriSC has a tool, called eTBc (easy TestBench creator), for automatic generation of partial testbenches. For each TDriver and TMonitor that compose the testbench, eTBc generates automatically the SystemC code to deal with transactions. Therefore, this phase does not require an extra work.

In the second phase, the engineer selects the attributes that must be analyzed. This part is equivalent to define the matrices that are used during the traditional cross product coverage analysis. Next, the engineer collects values from the database that was filled during the simulation. Considering the design of a simple calculator as example, that engineer can

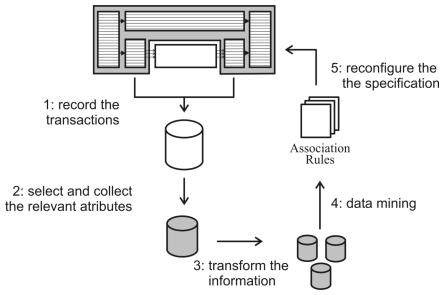

Figura B.2: The process to use association rules for analysis of cross product coverage.

extract the information about the operations that are executed in its ports. Furthermore, during the collecting sub-phase, the information is saved in a suitable format with regard the data-mining tool. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of sampled values during the simulation called items, for example, $I = \{command = add, command = sub, \ldots, port = 1, port = 2, \ldots, response = noResponse, response = successful, \ldots\}$. We are constructing the database D, $D = \{t_1, t_2, \ldots, t_m\}$ where each transaction t_i has a unique transaction ID and contains a non-empty subset of the items in I, called itemset. We are developing a tool support this activity.

The next phases are the data transformation and data mining. During the data transformation phase, the engineer performs fine adjustments to allow the mining of association rules. The tools that we are using for both phases are **R** [8] and the *arules* package [47]. An association rule is a pattern $X \rightarrow Y$ such that X are Y are sets of values. *Support* and *confidence* are key concepts for the data mining phase. The support of an association rule is the percentage of transactions in D that contain all of the items listed in that association rule. The confidence of a rule $X \rightarrow Y$ is defined as $conf(X \rightarrow Y) = supp(X \cup Y)/supp(X)$.

The number of possible association rules in I grows exponentially with number of elements in I. However, by setting expected values for support and confidence, we can filter the association rules and check if the combinations of values that were used to simulate the system are satisfactory. By using matrices in traditional CPC analysis, all that we have is the absolute values of how many times certain crossed items occurred. As support and confidence are not absolute values, they are more useful to help the engineer to adjust the stimuli generator. For example, the engineer can detect that operation *add* are rarely executed in *port 1*. Next, he can reconfigure the stimuli generator to fix this problem.

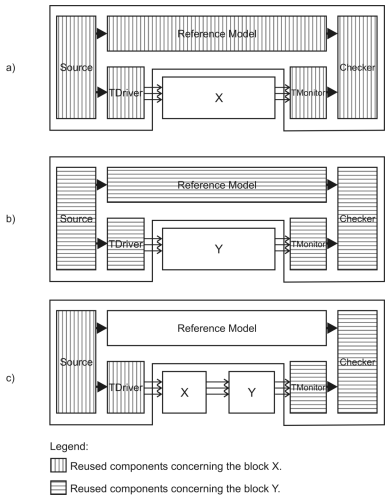

Figura B.3: Examples of testbenches: a) Testbench for the block X; b) Testbench for the block Y; c) Testbench for the composition Y ∘ X .

B.4 Improving Cross-Coverage Models

In a hierarchical design, the composition phase is performed combining one pair of blocks per time. The engineer can combine more than two blocks per time, but this is not recommended because when an error is observed, the cost to locate it is bigger. Thus, we are considering the composition in which one pair is combined per time. The functional verification of these blocks, according to the VeriSC methodology, requires one testbench for each block (Figure B.3, items a and b). The composition of both will produce a testbench in which the components Source and TDriver concerning the block X will be reused (Figure B.3, item c). Following the same principle, the components TMonitor and Checker will be reused from the block Y. As a result, the quality of the functional verification of the composition will be given by these reused components. The component TMonitor concerning the testbench of the block X as well as the components Source and TDriver of the block Y are disregarded.

The components being disregarded during the composition can be useful to evaluate if each block was satisfactorily verified stand-alone. We can perform the cross coverage analysis in the output of the block X and in the input of the block Y. By comparing the scores of the block X and block Y, we can judge if Y was verified satisfactorily with regard the block X. We perform this comparison by using values of support and confidence that were

obtained during the mining phase.

The original purpose of association rules mining is to detect regularities between products in market basket of consumers. Such information is used support decisions about marketing activities, for example, promotional pricing. Thus, the search for association rules with high support and high confidence is the focus. However, in our case, the focus is the search for low support and low confidence because the absence or low frequency of certain combinations can lead to uncovered areas in the design. Uncovered areas are undesirable because they can represent a behavior that is not in accordance with the specification.

B.5 Experimental Results

We analyzed the improvement provided by our technique by using it to verify two design blocks that compose the MPEG 4 video decoder. These blocks are ACDCIP and IQ, which are blocks for the inverse prediction and inverse quantization of frames respectively. This MPEG 4 video decoder is an OCP-IP compliant open-source SystemC-RTL that was implemented in a silicon chip. It has $22.7mm^2$ at a 0.35μm CMOS 4 ML technology with a 25 MHz working frequency [69]. It was originally developed using VeriSC and XVID software [82] as Reference Model.

We considered the hierarchical composition of ACDCIP and IQ blocks with other block, called BitStream. As BitStream provides data for both blocks, we can mine association rules in the output data of BitStream and compare these association rules with others that can be mined in the input data of ACDCIP and IQ. Instead of using the original testbenches, we injected errors in the specifications of stimuli generation and functional coverage concerning the blocks ACDCIP and IQ. With these *bad* specifications in hands, we applied our technique to improve them.

Next, we used two models of structural coverage to quantify the effectiveness our improvement: line coverage and branch coverage. We performed two batteries of verification, the first one using the bad specifications and the second one using the specification that were improved by our technique. During these batteries, we measured the line coverage and branch coverage for the RTL designs and for some blocks that compose the testbenches. To measure the structural coverage, we used the tool `gcov`, which work in conjunction with the `gcc`.

Table B.1 contains the information about our experiments. The information concerning the first battery of functional verification is presented in the second and fourth columns. Third and fifth columns contain the information about the second battery. By applying our approach, it was possible to improve the specification of both blocks. Hence, the line and branch coverage for the DUV of both blocks were improved (see Table 1, lines 1 and 2). The scores that were obtained by using the improved specifications are the same that were obtained using the original specifications.

	Line Coverage (%)		**Branch Coverage (%)**	
Module	**Old Spec.**	**New Spec.**	**Old Spec.**	**New Spec.**
acdcip.h	91.30	97.83	72.59	82.87
iq.h	77.71	78.93	48.42	48.91
tb.cpp	97.67	97.67	25.32	25.32
ac_ram.h	100.00	100.00	41.38	41.38

Tabela B.1: Structural coverage information about some blocks that compose the MPEG 4 video decoder.

B.6 Final Remarks

We presented an alternative for the use of matrices to perform CPC analysis during the functional verification of hardware designs. We performed CPC analysis by exploiting the techniques of association rules mining, originally developed for discovering interesting relations between variables in large databases. We used association rules mining to enhance the stimuli generation and coverage specification during the hierarchical composition of design blocks.