Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Geração Automática de Casos de Teste CSP Orientada por Propósitos

Sidney de Carvalho Nogueira

DISSERTAÇÃO DE MESTRADO

Recife
23 de agosto de 2006

Universidade Federal de Pernambuco
Centro de Informática

Sidney de Carvalho Nogueira

Geração Automática de Casos de Teste CSP Orientada por Propósitos

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Engenharia de Software.

Orientador: *Augusto Cesar Alves de Sampaio*
Co-orientador: *Alexandre Cabral Mota*

Recife
23 de agosto de 2006

Dedico está dissertação ao espírito do meu pai biológico, amigo eterno, conselheiro e ídolo, o Amado Guerreiro Anisio Pinto Nogueira (1946-2006), cujo exemplo de coragem, disciplina, perserverança, honestidade, empreendimento, desapego, amizade, alegria e bondade fortaleceram minha fé para concluir esta dissertação.

Agradecimentos

Agradeço :

À Deus, Nossa Senhora de Fátima, Jesus, Bezerra de Menezes (espírito) e André Luis (espírito).

À minha família, mais especificamente Sulamita (minha mãe), Alamita (irmã mais velha), Jacqueline (irmã mais nova) e Alexandre (meu cunhado) pelo fundamental apoio e confiança.

À minha noiva Mariane pela doação constante de ânimo e incentivo ao longo deste Mestrado.

Aos meu Orientadores Augusto Sampaio e Alexandre Mota pela paciência, dedicação, sensibilidade, incentivo, orientação, revisão, contribuições e observações.

Ao Colega André Figueiredo da UFCG que não se furtou em fornecer material e esclarecimentos a cerca da Teoria de Teste de Tretmans.

À minha Colega de Mestrado Patrícia Muniz pelas dicas e o apoio referentes à escrita.

Ao Professor Jim Woodcock (University of York) pela contribuição a cerca do processo de obtenção de múltiplos traces para um refinamento.

À amiga Luzinete Beltrão pelo apoio psicológico.

Ao CNPQ que me deu o incentivo financeiro através da bolsa de estudos.

À Motorola Brasil, que através do projeto de pesquisa CINBTC-RD em colaboração com a UFPE, pôde oferecer um ambiente real para experimentações de estudo de caso, como também pela bolsa de estudos oferecida como incentivo ao fim do período da escrita.

Fé inabalável só é a que pode
encarar frente a frente a razão,
em todas as épocas da humanidade.

— ALLAN KARDEC (O Evangelho Segundo o Espiritismo,1864)

Resumo

O processo de desenvolvimento de software está sujeito a inserção de erros diversos cuja presença compromete a qualidade final dos produtos de software. Teste é uma atividade dinâmica e bastante custosa dentro das várias empregadas pela Garantia da Qualidade de Software. O objetivo de teste é demonstrar que um comportamento específico (cenário) de um sistema foi bem (passou no teste) ou mal sucedido (falhou no teste), através de um veredito.

Automação de testes visa tornar o processo mais ágil em atividades repetitivas, menos suscetível a erros humanos. Existem várias abordagens de geração automática de teste, baseadas na representação formal do comportamento do sistema, que empregam diferentes critérios de seleção para os testes. Quando o objetivo do teste é focar na investigação de certas propriedades ou comportamentos importantes do sistema a ser testado, podemos utilizar o critério de seleção denominado propósito de teste (*test purposes*).

CSP (Communicating Sequential Processes) é uma notação formal bastante expressiva, é uma álgebra de processos útil para especificar comportamentos de sistemas concorrentes e distribuídos, de hardware e software. Infelizmente, não parece existir na literatura, abordagens para geração de testes diretamente a partir de álgebras de processo como CSP. As abordagens existentes utilizam a representação operacional (sistemas de transições rotuladas — LTS) dos processos CSP.

O objetivo deste trabalho é introduzir uma estratégia para geração automática de testes consistentes (*sound*), elaborada inteiramente a partir da semântica denotacional de CSP (notação de processos e modelos semânticos). É definida uma teoria de testes baseada na Teoria de Testes de Tretmans. Um ponto comum entre estas teorias é que o conjunto de ações de entrada e saída para especificações (alfabetos), implementações e testes são separados, de forma a definir com precisão os vereditos para execução dos testes e a relação de conformidade entre implementação e especificação. Adicionalmente, uma relação de conformidade denotada **cspioco** é introduzida em termos de refinamentos de processos para determinar se o processo que representa a implementação a ser testada está coerente com o comportamento do processo da especificação. É apresentada, ainda, a estrutura e a utilização de uma ferramenta implementada com o propósito de avaliar esta abordagem dentro do ambiente de teste de um projeto de pesquisa que envolve uma cooperação entre o CIn-UFPE e a Motorola Industrial Limitada. Alguns experimentos práticos foram realizados neste contexto.

Palavras-chave: Geração de Testes, CSP, Propósito de Testes, **cspioco** , Checagem de Refinamentos

Abstract

The process of software development is subject to the insertion of errors of diverse types, whose presence compromise the final quality of the software products. Test is one of many techniques supplied by the Software Quality Guarantee, which is dynamic and involves a considerable cost to be performed successfully. The goal of testing is to demonstrate that a specific behavior (scene) of a system succeeded accordingly (passed) or not (failed), based on a verdict.

Test automation aims to improve the process of testing particularly in repetitive activities, as well as minimise human errors. There are various approaches to generate tests automatically based in the formal representation of the behavior of the system, which use different selection criteria for the tests. When the objective of the test is to focus in certain properties or important behaviors of the system under test we can use selection criteria called test purposes.

CSP (Communicating Sequential Processes) is an expressive formal notation. It is a process algebra designed to specify behaviors of concurrent and distributed systems (hardware and software). Unfortunately, we cannot find approaches reported in the literature about the generation of tests directly from process algebras such as CSP. The existing approaches use the operational representation (Labelled Transition Systems — LTS) of CSP processes.

The goal of this work is to introduce a strategy for the automatic generation of sound tests, elaborated entirely from the denotacional semantics of CSP (semantic notation of processes and models). We define a theory of tests based on the Theory of Tests of Tretmans. A commonality between these theories is that the entry and exit actions of the specifications (alphabets), implementations and tests are separate. This is an interesting way to define with precision the verdicts for the execution of the tests and the relation of conformity between implementation and specification. Additionally, we define a conformity relation called **cspioco** in terms of processes refinements to determine whether the process which represents the implementation under test satisfies the behavior described by the process of the specification. Furthermore, we present the structure and use of a tool which we have developed with the goal to evaluate the proposed approach in a research project cooperation that involves CIn-UFPE and Motorola Inc. Some practical experiments had been carried out using this context.

Keywords: Test Generation, CSP, Test Purpose, **cspioco** , Refinement Checking

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O processo de desenvolvimento de software está sujeito à inserção de erros diversos [GM04] cuja presença diminui a qualidade final dos produtos de software. Entretanto, a qualidade do software não pode ser verificada apenas depois do produto já pronto; existem muitas atividades relacionadas com a Garantia da Qualidade de Software, dentre elas teste. Teste é uma atividade dinâmica cujo objetivo é demonstrar que um comportamento específico (cenário) de um sistema foi bem (passou no teste) ou mal sucedido (falhou no teste), através de um veredito. É uma tarefa de trabalho intensivo, podendo alcançar cerca de 50% do custo total do desenvolvimento do software [GM04].

Um processo de teste consiste num conjunto de atividades, papéis e artefatos para avaliar a qualidade do produto testado. Os principais problemas deste processo são [BBC+03]: alto custo, seleção de bons casos de teste e sua automação. Existem várias técnicas comportamentais e estruturais que visam aumentar a manutenabilidade, configurabilidade e reusabilidade dos testes, gerar testes precisos de forma eficiente e efetiva, etc. Dentre estas técnicas destaca-se a automação.

Automação de testes visa tornar o processo mais ágil em atividades repetitivas, menos suscetível a erros humanos, mais fácil de se reproduzir e menos dependente da interpretação humana. Em [Que99], mostra-se que a automação de várias atividades do processo de testes provoca uma diminuição média de 75% do esforço total, em comparação com o esforço das mesmas atividades realizadas manualmente.

Um ponto de partida para examinar a relação entre métodos formais e testes é considerar que ambos são baseados em modelos (*model-based*). Notações formais podem fornecer modelos não ambíguos dos requisitos do sistema. Tais modelos podem ser processados por ferramentas que automatizam várias atividades do ciclo de testes. Utilizando métodos formais e teste em conjunto, pode-se reduzir o custo de implementação pela aplicação de técnicas de teste com maior antecedência no ciclo de vida, e aumentar o nível de automação do processo de teste.

Existem várias abordagens de geração automática de teste que são baseadas na representação formal do comportamento do sistema. Tais abordagens partem de uma especificação formal do sistema e produzem como saída um conjunto de casos de testes formais, onde os elementos tradicionais de uma caso de teste (procedimentos e vereditos) são escritos em alguma notação formal. Um dos benefícios de utilizar casos de teste formais é a possibilidade de garantir propriedades sobre os testes gerados e de poderem ser mecanicamente traduzidos para alguma outra notação mais adequada para a execução manual ou automática dos testes sobre uma implementação.

A partir de um modelo formal, várias técnicas de geração automática (específicas para o formalismo utilizado) podem ser empregadas para se obter um conjunto de testes. As possibili-

dades de combinação entre comportamentos e dados especificados são capazes de produzir uma quantidade muito grande ou mesmo infinita de testes. Conseqüentemente, critérios de seleção são empregados para dirigir a seleção de testes de forma a tornar viável a execução do conjunto de testes (dos pontos de vista qualitativo e quantitativo) sobre uma implementação. Quando o objetivo do teste é focar na verificação de propriedades ou comportamentos específicos do sistema a ser testado, podemos utilizar o critério de seleção denominado propósito de teste (*test purposes*). Um propósito de teste pode descrever fluxo de controle, fluxo de sinais, fluxo de dados, tempo e probabilidade para especificar as propriedades da especificação que se deseja presentes no conjunto de testes obtido. Sua definição é um processo intuitivo e usualmente realizado por humanos.

Técnicas existentes para geração automática de testes (a partir de notações formais e propósitos de teste) [Tre96c, Tre96b, LY96], usualmente utilizam notações operacionais, cuja característica é ter pouca expressividade e abstração. A especificação do comportamento dos propósitos de teste e casos de teste gerados são descritos e entendidos em termos operacionais. Como exemplo de formalismos, pode-se destacar os sistemas de transições rotuladas (*Labelled Transition Systems* — LTS) [Wik06d] e as máquinas de estado finitas (*Finite State Machines* — FSM) [Wik06a]. Tais notações possuem algumas desvantagens que dificultam o trabalho do engenheiro encarregado da tarefa de especificar o comportamento do sistema e de definir os propósitos de teste. Dentre elas, podemos citar:

- Pouca modularidade. Apesar de possuirem uma notação gráfica, representações operacionais como LTS e FSM são blocos monolíticos de especificação, com pouco ou nenhum nível de modularidade. Esta deficiência se deve ao fato que não são dotadas de operadores que permitem com naturalidade compor os vários blocos de especificação. Como conseqüência, escrever grandes especificações como um só bloco pode tornar-se uma tarefa árdua que propicia a ocorrência de erros; e,

- Ausência de suporte para definir dados abstratos. Notações operacionais não possuem suporte para a especificação de dados de forma abstrata. Estruturas primitivas como conjuntos e seqüências, que quando combinadas podem modelar estruturas de dados abstratas, facilitam a tarefa de definir os valores de dados para a especificação, evitando o esforço exaustivo de enumerar os valores de dados em cada comportamento especificado de forma desestruturada.

CSP (Communicating Sequential Processes) [Hoa85, RHB97] é uma álgebra de processos que pode representar máquinas de estado e seus dados de uma maneira abstrata e sucinta. CSP é uma notação útil para especificar e projetar comportamentos de sistemas concorrentes e distribuídos de hardware e software, contando com modelos semânticos capazes de representar o comportamento de formas variadas, pelas quais ferramentas podem verificar propriedades das especificações, tais como: ausência de *deadlock*, ausência de *livelock*, comportamento determinístico e refinamentos entre processos.

Infelizmente, as abordagens existentes para geração de testes são restritas, essencialmente, a modelos operacionais como LTS, que sofrem dos problemas relatados anteriormente.

Considerando que CSP é o formalismo padrão adotado em muitos contextos de utilização, mais particularmente dentro do projeto de pesquisa, CINBTC-RD, do Centro Brasileiro de

testes da Motorola (BTC) que envolve colaboração do Centro de Informática da UFPE, surgiu a oportunidade de se investigar o uso de CSP para geração dos testes. Apesar de ser possível, via conversão, obter um LTS que representa o comportamento de um processo CSP, e, portanto usar este LTS como entrada para uma das ferramentas de geração existentes (que geram como saída casos de teste em LTS) seria necessário converter de volta para CSP estes casos de teste. Portanto, teríamos duas conversões, uma de LTS para CSP (especificação) e outra de CSP para LTS (testes). Este cenário de utilização além de potencialmente ineficiente, devido ao esforço de conversão, precisaria estar associado a uma prova formal de que tais conversões preservam a semântica de CSP. Por todos esses motivos, somos encorajados a buscar uma abordagem de geração que utilize puramente CSP como o formalismo padrão para as entradas e saídas dos testes.

O objetivo deste trabalho é introduzir uma abordagem de geração automática de testes consistentes (*sound*) elaborada inteiramente com a semântica denotacional de CSP (notação de processos e modelos semânticos). A especificação de entrada, a definição de propósitos de teste e os casos de teste gerados utilizam notação, operadores e ferramentas de CSP. Definimos uma teoria de testes baseada na Teoria de Testes de Tretmans [Tre96c]. Um ponto comum entre estas teorias é que o conjunto de ações de entrada e saída para especificações (alfabetos), implementações e testes são separados, de forma a definir com precisão os vereditos para execução dos testes e a relação de conformidade entre implementação e especificação. Além disso, uma relação de conformidade denotada **cspioco** é introduzida em termos de refinamento de processos para definir se o processo que representa a implementação a ser testada está coerente com o comportamento do processo da especificação. Finalmente, é descrita a estrutura e a utilização de uma ferramenta implementada com o propósito de avaliar esta abordagem dentro do ambiente de teste do projeto BTC, onde alguns experimentos praticos foram realizados.

1.1 Organização da Dissertação

O Capítulo 2 fornece uma introdução sobre teste de software, apresentando as definições básicas e a relação de Testes com Garantia da Qualidade de Software. Este capítulo também caracteriza as diversas fases de teste, as técnicas, as abordagens e as estratégias mais comuns. Uma visão geral sobre os benefícios de se automatizar as atividades de teste, bem como uma listagem dos diversos tipos de ferramentas de suporte para automação, é descrita ao fim do capítulo.

O Capítulo 3 introduz o formalismo CSP. Isto é, os diversos operadores utilizados para definir processos, uma visão sobre a semântica denotacional e operacional, mostrando suas relações através de exemplos. E também uma intuição sobre os três modelos semânticos de CSP, que são usados para estabelecer as relações de refinamento entre os processos, com destaque no modelo de traces, descrito mais detalhadamente.

O Capítulo 4 mostra a combinação de métodos formais com testes (denominada de testes formais) e descreve a estrutura que relaciona métodos formais e testes, definindo o balanceamento requerido entre a necessidade de provar hipóteses e executar testes. Também se apresenta como usar hipóteses para selecionar casos de teste e como refiná-las. Uma visão geral sobre os vários tipos de critério de seleção de testes com destaque aos propósitos de teste é apresen-

tada, como também, a visão de teste formal baseado em diferentes formalismos (Baseados em Modelos, Baseado em Máquinas de Estado, Baseado em Álgebras de Processo e Baseado em LTS) com ênfase nos teste formais baseados em LTS (a partir deste advêm a base conceitual utilizada para o desenvolvimento deste trabalho). E finalmente, os conceitos de teste formal, execução formal e as várias relações de implementação existentes, caracterizando e listando várias ferramentas de geração de teste baseadas em modelos formais de comportamento.

O Capítulo 5 representa a principal contribuição deste trabalho. Partindo de uma técnica existente para validação de cenários em CSP, desenvolve e explicita paulatinamente a abordagem de geração de casos de teste CSP a partir de propósitos (também em CSP). Neste ponto, é introduzido o conceito de propósito de teste CSP, os fundamentos da seleção de cenários de teste via esse propósitos (formados a partir de processos primitivos que podem ser combinados e estendidos para realizar operações de seleção das mais simples às mais elaboradas). Em seguida, é definida e mostrada a relação de conformidade **cspioco** . Por fim, o capítulo elabora, em termos de refinamentos de CSP, conceitos como: execução, vereditos e a propriedade de consistência (*soundness*) para testes, mostrando como construir a partir de um cenário de teste um caso de teste CSP que é consistente (*sound*).

O Capítulo 6 mostra a ferramenta ATG que implementa a abordagem de geração do Capítulo 5 e o estudo de caso realizado. É introduzido um formato simplificado para definição de propósitos de teste (pseudo propósitos) fornecido com uma das entradas da ferramenta para que sejam gerados os casos de teste CSP. Em acréscimo, é descrito o ambiente, e as adaptações realizadas no mesmo para que fosse possível realizar um estudo de caso com a ferramenta, descrevendo o caso selecionado, a metodologia empregada bem como os resultados alcançados.

Finalmente, no Capítulo 7 são apresentadas as conclusões sobre a abordagem construída, a utilização da ferramenta em experimentos, os trabalhos relacionados e os trabalhos futuros.

CAPÍTULO 2

Teste de Software

O desenvolvimento de software está sujeito a inserção de erros diversos [GM04]: erros na especificação, erros no momento de detalhar e definir uma arquitetura para o sistema e os erros inseridos durante a implementação do sistema.

De acordo com o glossário de software da IEEE [Com98], erro é um sinônimo para a falha humana na execução de uma atividade de desenvolvimento de software; quando o erro ocorre durante a codificação é chamado de bug. Falta é o resultado de um erro, ou a sua representação, e pode ocorrer em diferentes artefatos ao longo do processo. Falha ocorre quando uma falta é executada.

A prevenção e correção de erros é um dos objetivos centrais da Garantia da Qualidade de Software (GQS) que abrange todo processo de desenvolvimento: desde especificação até a manutenção do sistema. [How06]. Entretanto, qualidade de software não atua apenas quando o produto já estiver pronto.

As técnicas de VV&T — Validação, Verificação e Teste [Lew00] — estão diretamente relacionadas com a Garantia da Qualidade de Software. Validação consiste em assegurar que os requisitos do software estão de acordo com a real intenção do usuário (Estamos construindo o produto certo?). Já verificação, objetiva assegurar consistência, completude e corretude do produto em cada fase e entre fases consecutivas do ciclo de vida do software (Estamos construindo corretamente o produto?). E finalmente, teste que visa examinar o comportamento do produto em cenários específicos executando a implementação real do sistema (geralmente seguindo algum processo de teste). Teste tem propósito duplo: demonstrar a presença de falhas nos itens de teste bem como o comportamento esperado (correto) para os cenários específicos pela execução. Neste trabalho, a implementação será denominada pela sigla IUT (Implementation Under Test).

Um processo de teste consiste num conjunto de atividades, papéis e artefatos para avaliar a qualidade do produto testado. As atividades mais essenciais são projetar testes e executar testes. Essas atividades são desempenhadas pelos papeis, respectivamente, do projetista de teste (*test designer*) e do testador (*tester*). O projetista gera como artefatos a especificação de casos de teste e a especificação dos procedimentos de teste. A seguir, de acordo com o padrão IEEE [Com98] tem-se as seguintes definições para os artefatos produzidos pelo projetista de testes.

Item de teste (Test item) Um código-fonte, código objeto, dados de controle, etc; o objeto a ser testado.

Especificação de caso de teste (Test case specification) Um documento especificando entradas, resultados esperados, e um conjunto de condições de execução para um item de teste.

Especificação de Procedimento de teste (Test procedure specification) Um documento especificando a seqüência de ações para a execução de um teste.

De forma mais comum na literatura, é denominado teste (artefato que o testador utiliza como entrada) um conjunto de um ou mais casos de teste, ou, um conjunto de um ou mais procedimentos de teste, ou, um conjunto de um ou mais casos e procedimentos de teste. Para saber se o teste passou ou não faz-se uso do termo oráculo, que é um mecanismo (humano - próprio testador, ou, automático - uma ferramenta) que, considerando as entradas fornecidas no teste, saídas produzidas pela IUT e saídas esperadas, define o veredito da execução (passou/falhou). Passou significa que a IUT apresentou o comportamento especificado e falhou o caso contrário (o que caracteriza uma falha).

Infelizmente, teste de software tem algumas desvantagens: É uma tarefa de trabalho intensivo podendo alcançar a marcar de até 50% do custo total do desenvolvimento do software [GM04], não garante a ausência completa de falhas e ainda é foco de introdução de erros, o que pode tornar os resultados não confiáveis. Em relação a esta última desvantagem, os erros podem acontecer da má especificação dos casos de teste e também da ausência de teste para requisitos funcionais e não-funcionais que são fundamentais.

Na literatura de testes (dependendo do autor) existe uma grande variação no emprego dos termos: fase de teste, nível de teste, etapas de teste, tipos de teste, abordagens de teste, técnicas de teste, etc. As Seções 2.1 e 2.3 introduzem a terminologia própria deste trabalho.

2.1 Níveis

Para ilustrar os vários níveis de teste, utilizamos o modelo de desenvolvimento em V [Som01] (V-Model) — derivado do modelo de desenvolvimento em cascata. Este modelo tem esse nome devido à ordem de ocorrência das atividades de desenvolvimento e teste, que pode ser vista na forma da letra V na Figura 2.1.

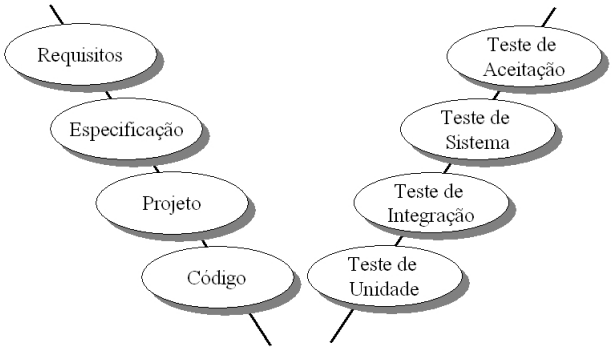

Figura 2.1 Modelo em V

À esquerda da Figura 2.1 estão as fases de desenvolvimento e à direita os diferentes níveis do processo de testes. Começando o processo pelos requisitos, fase superior esquerda da Figura

2.1, continuamos a avançar descendo fase por fase até alcançar a fase mais abaixo do processo de desenvolvimento que é a obtenção do código. Nesse ponto, as atividades de teste se iniciam, começando no nível de teste unitário e movendo para cima um nível de teste por vez até alcançar o teste de aceitação, esse último localizado no extremo superior direito da Figura 2.1. A seguir apresentamos a terminologia para cada um do níveis de teste citados:

Teste de unidade Verifica o correto funcionamento de cada módulo ou componente do sistema em separado;

Teste de integração Verifica o correto funcionamento do conjunto de componentes do sistema à medida que eles são integrados;

Teste de sistema Após integrados os módulos e componentes, verifica-se o funcionamento do sistema dentro de um ambiente operacional de desenvolvimento;

Teste de aceitação Verifica o funcionamento do sistema no ambiente operacional do cliente.

O modelo em V ilustra bem o mapeamento das fases do desenvolvimento centrado no código com os níveis de teste. O código está para os testes de unidade, o projeto está para os teste de integração, a especificação está para o teste de sistema e o requisito está para o testes de aceitação. Nesse contexto, a verificação dos artefatos é realizada na ordem inversa com a qual eles são produzidos. O código é o primeiro a ser testado no entanto, o último a ser produzido; os requisitos são verificados por último, apesar de estarem prontos com maior antecedência.

Apesar de fácil de entender, o modelo em V não é uma abordagem adequada para os processos iterativos de software atuais, pois os testes apenas são iniciados depois que toda a implementação estar pronta. Tal modelo, apesar de ultrapassado, é útil quando esses mapeamentos são aplicados em um processo onde as diferentes etapas do desenvolvimento são realizadas e testadas simultaneamente em múltiplas iterações (por exemplo o RUP [Kru00]). Dessa forma, o modelo em V pode se tornar um modelo efetivo para um processo de teste gerenciável.

Dentro de cada um dos níveis acima são empregadas estratégias particulares para teste. Uma estratégia consiste, em determinado nível de teste, em montar uma combinação apropriada de abordagens de teste (caixa-branca, caixa-preta, etc) com os tipos de teste apropriados (funcionalidade, performance, interface, etc). Na Seção 2.3 são descritas abordagens e tipos de teste.

2.2 Metas para um processo de teste

Devido ao alto custo, existe uma necessidade crítica de prover suporte para o processo de teste que lide com os seguintes requisitos:

1. Aumentar a efetividade do teste: prover um suporte para identificar casos de teste que tenham a maiores chances de revelar falhas e incidentes. Um teste efetivo é aquele que encontra mais falhas;

2. Aumentar a eficiência do teste: identificar estratégias bem fundamentadas para reduzir o número de casos de teste que precisam ser executados sem impacto significante na efetividade global do conjunto de testes. Um teste eficiente pode substituir vários que não o são com relação a cobertura alcançada, quantidade de falhas encontradas, etc;

3. Geração precisa dos casos de teste: garantir ao mesmo tempo a cobertura e conformidade com os requisitos; e

4. Aumentar manutenibilidade, configurabilidade e reusabilidade dos casos de teste: para maximizar o retorno do esforço investido na escrita dos casos de teste.

Os principais problemas no processo de teste são: seleção de bons casos de teste (acompanhados de suas respectivas saídas e sua automação (que envolve controlar o processo de teste e observar os resultados dos testes). Na Seção 2.3 são mostradas algumas técnicas caixa-branca e caixa-preta empregadas para gerar casos de teste de tanto de maneira efetiva quanto eficiente (Metas 1 e 2). Na Seção 2.4 são mostradas ferramentas voltadas para automação de diversas atividades do processo de teste (Metas 3 e 4).

2.3 Abordagens

Quanto à abordagem, existem basicamente duas técnicas para projetar os testes caixa-preta (black box) e caixa-branca (white box).

Abordagens caixa-preta levam em conta apenas aspectos do comportamento do programa para projetar os casos de teste [Bei95]. São ignorados aspectos internos do software, como módulos e linguagem de programação. Os artefatos considerados são documentos de requisitos e especificações funcionais. São sinônimos para teste caixa-preta: teste funcional, teste de comportamento, teste de caixa-opaca, e teste caixa-fechada.

Como exemplo, seja uma IUT que possui um campo para entrada de valores inteiros que variam de -100 a 100. Para valores de entrada dentro desse limite, o sistema processa a entrada e responde com o quadrado do valor entrado. Para valores fora do limite especificado, o sistema informa o usuário de que o valor fornecido como entrada é incorreto. Utilizando como base esse exemplo, a seguir são apresentadas duas técnicas bastante conhecidas associadas à abordagem caixa-preta.

Análise do Valor Limite Técnica utilizada para identificar os principais limites de valores de dados onde mudanças significativas de comportamento do sistema testado podem ocorrer. A experiência tem mostrado que existe um alto risco de inserção de erros na programação das condições lógicas que definem os limites de comportamento dos produtos. Tendo identificado esses limites, são criados casos de teste que verificam o comportamento da IUT quando fornecidos valores próximos aos limites especificados. Supondo que estamos testando a IUT do exemplo acima, pode-se identificar 0 como valor central para entradas e, -100 e 100 os valores limite. Portanto, os testes desenvolvidos vão fornecer ao programa os seguintes valores de entrada: $-101, -100, -99, -1, 0, 1, 99, 100, 101$. Os valores -101, -99, 99 e 101 estão ao redor do limite e também são incluídos como

entradas para os testes. Esta é uma prática largamente utilizada e tida como estratégia efetiva para projetar casos de teste, uma vez que maximiza as chances de encontrar erros.

Particionamento de Equivalência Técnica que seleciona casos de teste representativos considerando um grande número de casos de teste que possuem comportamento semelhante. A hipótese chave para sua aplicação é que a IUT possui comportamento uniforme considerando entradas que estão dentro de uma mesma partição de valores. Nesse método são observadas as partições e selecionados um ou mais valores de cada partição como parâmetros de entrada fornecidos pelos casos de teste. Considerando que a IUT é o mesmo exemplo da técnica anterior, pode-se identificar três partições de valores: $[-100,0)$,$[0]$, $(0,100]$; donde se pode extrair (ao menos) um teste para cada uma das partições. Em princípio, esta é uma estratégia eficiente para projetar casos de teste, minimizando o número de casos de teste necessários que precisam ser executados para obter um dado nível de confidência da IUT. Esta é uma estratégia eficiente para construir casos de teste.

Na abordagem caixa-branca, o projetista de testes utiliza informações internas do software (estrutrura) para criar dados de teste a partir da lógica de programação. Sinônimos para teste caixa-branca incluem: teste estrutural, teste caixa de vidro e teste caixa clara. A seguir podemos ver algumas técnicas de teste caixa-branca:

Análise Estática (static analysis) Envolve percorrer o código-fonte (geralmente com auxílio de ferramentas) com o objetivo de encontrar potenciais defeitos que podem causar falhas durante a execução do programa. Por exemplo, através da análise estática do código-fonte C++ podem ser identificados pontos que provavelmente causarão estouro de alocação de memória.

Cobertura de comandos Nesse tipo de teste, o código é executado de uma maneira tal que cada comando da aplicação (program statement) é executado pelo menos uma vez.

Cobertura de condições Teste de cobertura de condições (branch coverage) ajuda a validar se todos as condições (if/them/else/switch) foram executadas pelo menos uma vez, garantindo que nenhuma das escolhas leva para um comportamento anormal.

Teste de segurança Tem por finalidade verificar quão bem a IUT pode proteger-se de acessos não autorizados, como exemplo, hackers, crackers e alterações de código perigosas que lidam com o código da aplicação.

Teste de mutação (Mutation Testing) Objetiva executar a IUT após a introdução artificial de defeitos dentro do código para analisar a efetividade dos casos de teste existentes em detectá-los.

Existe ainda uma combinação das duas primeiras abordagens, denominada caixa-cinza (grey box) ou caixa-translúcida (translucid-box); nesta abordagem são levados em conta tanto aspectos comportamentais tanto quanto estruturais [Lew00]. Isto é, o projetista de testes caixa-preta entra em contato com o desenvolvedor para obter informações sobre a estrutura da implementação. Essa informação será utilizada para tornar o processo de teste mais eficiente.

Como exemplo de caixa-cinza, suponha que o desenvolvedor informa ao engenheiro de testes que uma porção do software, relativa a uma funcionalidade, está sendo reusada de uma versão de software testada com sucesso anteriormente. A partir dessa informação, o engenheiro de testes pode eliminar, do conjunto de testes a ser executado, os testes relativos a funcionalidade já testada, cuja implementação é reaproveitada.

Teste de caixa-branca está associado apenas com teste do produto de software, isto é, não garante que a especificação seja totalmente implementada. Já o caixa-preta baseia-se apenas na especificação, não podendo garantir que todas as partes da implementação sejam testadas. Portanto, caixa-preta consiste em testar a IUT contra a especificação para descobrir faltas de omissão (omission faults), indicando que partes da especificação não são cobertas. Enquanto que o de caixa-branca consiste em testar fatores adicionais com relação à implementação e descobrir faltas de comissão (commission faults), indicando que parte da implementação está com defeito. Para testar de maneira completa um produto de software, é necessário utilizar a combinação de técnicas caixa-preta e caixa-branca. Uma estratégia recomendada em [BBC+03] é:

1. Usar técnicas de caixa-preta para identificar o conjunto inicial de casos de teste;
2. Usar algumas medidas conhecidas de cobertura de código para averiguar a cobertura inicial dos casos de teste;
3. Comedida e cuidadosamente adicionar casos de teste até que o nível de cobertura esteja de acordo com algum medida padrão da companhia.

De forma ortogonal às técnicas descritas aqui, existem tipos de teste que podem ser realizados independente das classificações anteriores: funcionalidade (verifica se a funcionalidade está como esperada), performance (verifica o tempo de resposta do sistema), carga (verifica a resposta do sistema dado a inserção variável de cargas), interface (verifica a interface do usuário), estresse (testa a recuperação do sistema em situações críticas onde existe escassez ou ausência de recursos), instalação (verifica se a instalação do software ocorre de maneira correta em diferentes ambientes operacionais), etc.

Maiores detalhes sobre as técnicas de teste podem ser encontrados em [Lew00]. Bem como exemplos de suas utilizações em [Jor95].

2.4 Automação

A automação das atividades visa tornar o processo [Que99] : mais ágil em atividades repetitivas, menos suscetível a erros humanos, mais fácil de reproduzir suas atividades e menos dependente da interpretação humana.

Alguns tipos de teste, como os testes de performance e estresse, apenas podem ser conduzidos de forma satisfatória quando automatizados. Nesses tipos de teste, a repetição excessiva dos procedimentos pode levar o testador ao cansaço e ao erro que comprometerá os resultados finais.

Alguns requisitos de performance ou estresse requerem uma quantidade de iteração impraticável para seres humanos, por exemplo, simular o acesso simultâneo de 1000 usuários ao sistema. Ferramentas de automação, como [Apa06b], podem produzir a taxa de acesso que simula uma taxa de acesso humanamente impraticável. Além disso, testes automáticos podem produzir métricas de qualidade e permitir a otimização dos testes.

O processo de automação pode ser medido e repetido com maior facilidade, pois as ferramentas podem armazenar os resultados da execução em logs para uma análise posterior de um Engenheiro de Qualidade de Software (SQE — Software Quality Engineer) que irá proceder a análise de qualidade através das seguintes métricas: quantidade de testes que passaram e falharam, tempo total e médio da execução do conjunto de testes, cobertura dos testes, etc.; entre outros fatos, as métricas fornecidas podem ser utilizadas para motivar a automação dos testes.

O uso de ferramentas de teste, como scripts de teste, pode aumentar a profundidade e a quantidade de testes executados. Scripts são pequenos programas escritos em uma linguagem de script para testar certo conjunto de funcionalidades da IUT. Podem reproduzir os testes automaticamente, o que torna viável executar o mesmo conjunto de testes sobre diferentes tipos de hardware e configurações de ambiente de uma mesma versão da IUT. Isso pode melhorar a qualidade dos testes, com a garantia de que as mudanças de hardware não afetam o comportamento da IUT. Testes automáticos permitem que a maior parte das funcionalidades básicas da interface do usuário possam ser cobertas de forma que a IUT seja considerada madura o suficiente para seguir para outras etapas de teste que focam nas regras de negócio.

Na Tabela 2.1 podemos ver uma pequena amostra das diversas ferramentas de teste existentes, cada com com seu propósito específico.

Propósito da Automação	**Nome da Ferramenta**
Planejamento e projeto de testes	Rational TestManager [Sof99], QADirector [Com06a]
Geração de massas de dados	File-Aid [Com06b]
API de testes unidade e Verificação de assertivas	JUnit [JUn06] e Cactus [Jak06]
Testes de cobertura	PureCoverage [Rat06]
Inspeção e análise estática de código	JTest [Par06], IntelliJ IDE [Int06]
Testes de GUI	Rational Robot [IBM06], WinRunner [Mer06b]
Testes de carga e estresse	JMeter [Apa06b], LoadRunner [Mer06a]
Testes de desempenho e detecção de gargalos	JProbe [Sof06]

Tabela 2.1 Ferramentas de Automação de Teste

Apesar de ser uma solução lógica para vários contextos de utilização, ainda existe um grande mito ao redor dos benefícios da automação de testes. Há uma enorme expectativa que nem sempre pode ser satisfeita. Muitos pensam que a automação, utilizada indiscrimi-

nadamente, pode resolver todos os problemas relacionados a testes, melhorar a qualidade do software, diminuir custos, e eliminar a interação humana. No entanto, a inserção de automação acrescenta novos custos e requer conhecimento adequado para uma utilização comedida de ferramentas, levando em conta os custos e benefícios associados.

Uma vez adotadas ferramentas para automação de testes, os resultados não aparecem instantaneamente [Que99]. De fato, existe um aumento de esforço para criar os scripts automáticos em contraste com os testes manuais. O próprio tempo para preparar o ambiente e iniciar a executar automática dos testes é maior. Existe uma curva de aprendizagem, e até que a etapa inicial seja superada, espera-se uma diminuição da produtividade durante o uso inicial da automação.

Estudo relatado em [Que99], considera que o esforço empregado nas atividades de planejamento de testes, projeto de casos de teste, criação de relatórios de teste e análise de relatórios de teste; provoca uma diminuição média de 75% de esforço total quando é utilizada automação de teste em um processo que antes era manual. Pode-se destacar as atividades projeto de testes e de execução de testes como as que obtiveram uma redução mais significativa, 55 % e 95 %, respectivamente.

Felizmente, de acordo com [PPW$^+$05] que, compara a capacidade de um conjunto de testes detectar erros de especificação e implementação, foi constatado que, testes gerados automaticamente a partir de modelos formais, podem detectar uma quantidade maior de erros de especificação do que os testes gerados manualmente a partir dos mesmos requisitos.

Por conseguinte, no próximo capítulo apresentamos uma linguagem formal apropriada para descrever aspectos comportamentais e que será usada nos capítulos subseqüentes para dar origem aos casos de teste de forma automática.

Capítulo 3

CSP

CSP [Hoa85, RHB97] (Communicating Sequential Processes) é uma notação formal utilizada para descrever sistemas concorrentes onde os componentes interagem entre sí através de um mecanismo de comunicação. Pertence a uma classe de formalismo chamada álgebras de processo cuja formulação reside sobre um rico conjunto de leis algébricas e cálculos para verificação das propriedades. A versão de CSP que nos baseamos é a de Roscoe [RHB97], que implementa algumas modificações sobre a versão original de Hoare [Hoa85].

O processo é a abstração utilizada para especificar comportamento. São construídos através de eventos, operadores e outros processos. Um sistema pode ser modelado através de um ou mais processos independentes que podem ser combinados para formar novos processos mais complexos: cada processo pode ser utilizado como fração do comportamento.

Um evento é um abstração para a ocorrência de um fato real, é o objeto mais elementar de CSP. Por exemplo, o evento *button*.1 pode ser utilizado para representar a ação pressionar o botão número 1. Apesar do fato modelado levar algum tempo para iniciar e ser concluído, um evento representado em CSP ocorre instantaneamente: é dito que um evento simplesmente ocorreu, não existe a noção de tempo associada. No exemplo do botão, quando o evento *button*.1 é comunicado, interpreta-se que o botão simplesmente foi pressionado. A versão de CSP que será trabalhada é sem tempo (untimed), pois não especifica restrições de tempo sobre a ocorrência dos eventos: considera apenas a ordem em que os eventos ocorrem. Extensões de CSP com tempo (*Timed CSP*) [Sch92] já existem, porém não são consideradas neste trabalho.

Um evento pode pertencer a uma classe de valores denominada *canal*. Um canal representa uma coleção de eventos com características comuns. Como exemplo, temos a seguinte especificação:

$$channel\, button : Int$$

A especificação acima introduz o canal *button* que pode transmitir qualquer valor do tipo inteiro (*Int*); o evento *button*.1 é um dos possíveis eventos comunicados a partir dessa declaração. Um canal pode ser observado como o conjunto resultante da enumeração de todos os seus eventos, por exemplo, o canal *button* pode ser visto como o conjunto de eventos

$$EV = \{button.1, button.2, ..., button.n\}$$

onde *n* é o valor máximo para o tipo *Int* predefinido. O operador de expansão aplicado ao nome de um canal (| *nomeCanal* |), retorna um conjunto com a enumeração de todos os eventos do mesmo. Por exemplo, a expansão do canal | *button* | retorna o conjunto *EV*. É permitido definir os valores comunicados pelo canal em termos de tipos e estruturas definidos pelo usuário. Como exemplo, a seguinte especificação:

$MAX\_OPTIONS = 4$
$OPTIONS = 1...MAX\_OPTIONS$
$channel\, option : OPTIONS$

O operador de compreensão de conjunto permite definir conjuntos de forma implícita. Na especificação acima, o conjunto *OPTIONS* é definido contendo valores inteiros contíguos e crescentes que iniciam pelo valor 1 e vão até o valor definido pela constante *MAX_OPTIONS*. Como o valor desssa constante é 4, o conjunto *OPTIONS* é igual a:

$$\{option.1, option.2, option.3, option.4\}$$

Conseqüentemente, a expansão do canal *option* (| *option* |) equivale ao mesmo conjunto *OPTIONS*. Esses eventos especificam o pressionamento de um seletor com quatro opções: *option*.1, corresponde a seleção da primeira opção; *option*.2, corresponde a seleção da segunda opção e assim por diante para os outros eventos. A declaração de um canal sem tipo, como

$$channel\, turnoff$$

define um único evento *turnoff*, que pode ser utilizado para representar a ação de desligar um aparelho.

Eventos fazem parte de um alfabeto denotado pela letra Σ, que contém todas as possíveis comunicações para os processos dentro do universo considerado. Por exemplo, se um processo de nome *P* comunica os eventos *button*.1, *button*.2 e *button*.3, seu alfabeto será por convenção Σ_P (Σ acrescido do nome do processo subscrito) cujo valor é igual a $\{button.1, button.2, button.3\}$.

A ocorrência de um evento em um processo caracteriza uma comunicação deste processo com pelo menos um participante. Geralmente o participante é um outro processo, caso contrário será o próprio ambiente em que o processo está inserido. A comunicação entre processos é atômica e se dá através de passagem de mensagens simultâneas. Como o modelo de comunicação é síncrono, todos os processos participantes devem estar simultaneamente prontos para executar a comunicação. Por exemplo, se o evento *turnoff* é oferecido por um processo *P*, e *P* não está sincronizado com outro(s) processo(s), o processo estará sincronizado ao menos com seu ambiente. Nesse caso, o evento *turnoff* só será comunicado quando o ambiente sincronizar neste evento, enquanto isso não ocorre, *P* irá esperar indefinidamente até que o ambiente realize a sincronização e possa comunicar o evento.

A composição de eventos para formar um processo, e o relacionamento entre diferentes processos são descritos através dos operadores algébricos de CSP. A sintaxe de CSP define a forma como eventos e processos podem ser combinados, através dos operadores, para formar um processo. A seguir apresentamos a sintaxe considerada nesta trabalho [RHB97] seguida de uma breve descrição para cada um dos elementos sintáticos a serem utilizados.

3.1 Definição de Processos

Cada processo é uma *equação* composta por eventos, operadores algébricos e outros processos. As equações de processos são definidas de forma semelhante a funções. Ou seja, o lado

$P(s)$	::=	$Stop$	
	|	Skip	
	|	$a \rightarrow P$	(prefixo)
	|	$P(s)$	(recursão)
	|	$P(s) \,\square\, P(s)$	(escolha externa)
	|	$P(s) \sqcap P(s)$	(escolha interna)
	|	$if\,(g)\,then\,P(s)\,else\,P(s)$	(escolha condicional)
	|	$P(s) \setminus C$	(internalização)
	|	$P(s)\,;\,P(s)$	(composição seqüencial)
	|	$P(s) \,\triangle\, P(s)$	(interrupção)
	|	$P(s) \lvert[\, C \,]\rvert P(s)$	(paralelismo)

Tabela 3.1 Processos primitivos e operadores algébricos de CSP

esquerdo introduz o nome e um ou mais parâmetros, e o lado direito o corpo do processo. Como exemplo, assumindo que o lado esquerdo da equação de um processo parametrizado P será representado como $P(s)$, onde s é o parâmetro de P. Os parâmetros de processo, junto com os parâmetros de entrada de eventos, representam o estado interno do processo. O escopo dos parâmetros de processo se estende por toda a definição do processo, e seus valores podem ser utilizados em qualquer evento deste processo.

A Tabela 3.1 contém uma listagem dos operadores algébricos e os processos primitivos de CSP que serão utilizados para compor equações de processos mais elaborados. Nesta Figura, considere g uma expressão condicional (que retorna um booleano), a um evento ($a \in \Sigma$) e C um conjunto de eventos.

3.1.1 Processos Primitivos

Existem dois processos primitivos em CSP: *Stop* e *Skip*. O processo *Stop* representa um estado problemático de um sistema (o sistema quebrou), também pode ser usado para representar um *deadlock*. Em contra partida, o processo *Skip* representa o término de uma execução com sucesso.

Os processos *DIV* e *RUN* são bastante úteis entretanto, não são processo primitivos, desde que são construídos a partir de outros processos. O processo *DIV* representa um estado de *live-lock*, e pode ser simulado através de um processo que executa ações internas indefinidamente, e as ações não podem ser percebidas por outros processos ou pelo ambiente. O comportamento de $RUN(C)$ é sincronizar com qualquer evento que pertence ao conjunto C, e em seguida volta a comporta-se como $RUN(C)$ novamente, isto é, sincroniza em qualquer evento de C indefinidamente.

3.1.2 Prefixo

Para construir um processo através de seus eventos usamos o operador $\rightarrow$ (chamado operador de prefixo), ver Tabela 3.1. O operador de prefixo é utilizado com um evento do lado esquerdo e um processo do lado direito. Como exemplo, temos o comportamento do processo $S8$, que é

definido por um prefixo

$$S8 = y \rightarrow S0$$

este processo oferecer o evento y ao ambiente e aguardar indefinidamente até que o ambiente esteja pronto para sincronizar neste evento. Quando isso aconteçe, o processo passa a comporta-se como o processo $S0$. Também é possível criar um processo com vários prefixos em seqüência, como no exemplo do processo $S10$:

$$S10 = d \rightarrow y \rightarrow S4$$

O processo $S10$ aguarda que o ambiente esteja pronto para sincronizar no evento d, após a sincronização e comunicação desse evento se comporta como o processo $y \rightarrow S4$, este último aguarda a sincronização no evento y para em seguida se comportar como o processo $S4$. O aninhamento de prefixos, um prefixo definido por outro, pode ser empregado ilimitadamente.

3.1.3 Recursão

Quando se quer representar comportamentos cíclicos, utiliza-se o operador de prefixo combinado com um mecanismo adicional para representar comportamentos repetitivos. Tal mecanismo chama-se recursão. A recursão pode ser útil para definir um processo que é definido em termos de sí próprio, como o processo P da Tabela 3.1. Este processo oferece o evento a e espera que o ambiente sincronize, após a sincronização se comporta de acordo com a recursão P (como sí próprio indefinidamente). É também útil para definir processos que possuem recursão mútua entre si, como o exemplo a seguir.

$$S6 = y \rightarrow S7$$
$$S7 = c \rightarrow S6$$

O processo $S6$ oferece o evento y, aguarda a sincronização e se comporta com $S7$, este último por sua vez, oferece c e aguarda a sincronização, após a sincronização de comporta novamente como $S6$. Outro processo equivalente a esses dois é P2,

$$P2 = y \rightarrow c \rightarrow P2$$

que vai eferecer e sincronizar em y, depois em c, e volta a se comportar recursivamente como ele próprio.

3.1.4 Escolha Externa e Interna

É comum que um processo ofereça mais de um evento distinto ao mesmo tempo, e o próximo evento a ser sincronizado possa ser escolhido. Esse tipo de alternativa é especificada via operadores de escolha. O operador de escolha externa (denotado pelo símbolo $\Box$) possibilita que o ambiente no qual o processo está inserido realize essa escolha. Tomando como exemplo o processo $S2$ a seguir:

$$S2 = c \rightarrow S6 \Box b \rightarrow S4$$

Este processo oferece simultaneamente dois eventos : c do lado esquerdo da escolha e b do lado direito. Fica ao encargo do ambiente selecionar com qual dos eventos deseja tornar disponível para sincronização. Caso selecione sincronizar em c, o processo $S2$ comunica c e se comporta como o processo $S6$, caso contrário, se sincronizar em b, o processo $S2$ comunica b e se comporta como o processo $S4$. Escolha externa é também conhecida como operador de escolha determinística, pois todos os eventos a serem sincronizados são visíveis ao ambiente, que seleciona o evento oferecido.

Existe uma forma indexada para o operador de escolha externa.

$$\Box\, ev : C @ ev \rightarrow P$$

A especificação acima corresponde a escolha externa indexada para os eventos $ev \in C$; possui o seguinte comportamento:

$$ev1 \rightarrow P \,\Box\, ev2 \rightarrow P \,\Box\, ... \,\Box\, evx \rightarrow P$$

Onde os eventos $ev1$, $ev2$ e evx são todos eventos que pertencem ao conjunto C. Dessa versão indexada podemos definir o processo $RUN(C)$ da Seção 3.1.1.

$$RUN(C) = \Box\, ev : C @ ev \rightarrow RUN(C)$$

O operador de escolha interna (denotado pelo símbolo $\sqcap$) é utilizado quando se quer deixar a encargo do próprio processo escolher quais eventos serão disponibilizados para sincronização com o ambiente. Com esse operador, o próximo evento oferecido é uma decisão interna do processo, da qual o ambiente não tem controle. O comportamento do processo $S2'$

$$S2' = c \rightarrow S6 \sqcap b \rightarrow S4$$

é oferecer, ou recusar, para sincronização, tanto o evento c quanto o evento b. Como isso ocorre de forma imprevisível, esse operador também e conhecido como operador de escolha não-determinística.

Pode-se obter comportamentos não-determinístico, mesmo utilizando o operador de escolha externa. Isso pode ocorrer quando são oferecido eventos iguais dentro da mesma escolha externa. O processo

$$S0 = t \rightarrow S9 \,\Box\, t \rightarrow S2$$

especifica um escolha externa entre dois eventos idênticos t. Dependendo da escolha de qual t seja oferecido e sincronizado, o processo $S0$ de comporta como $S9$ ou $S2$. Essa escolha será realizada internamente por $S0$, pois o ambiente não tem condições de escolher qual dos eventos t será escolhido, pois são idênticos. Como conseqüência, $S0$ é não-determinístico. O processo

$$S0' = t \rightarrow S9 \sqcap t \rightarrow S2$$

que emprega escolha interna, é equivalente a $S0$ que emprega escolhas externas.

Comportamentos não-determinísticos em geral são situações indesejáveis em sistemas complexos e concorrentes, porque representam imprevisibilidade ou falhas de modelagem. Entretanto, o operador $\sqcap$ (escolha interna) pode ser usado para abstrair detalhes internos do comportamento de processos, como condições de controle não modeladas.

3.1.5 Escolha Condicional

Além das escolhas (interna e externa) apresentadas na subseção anterior, CSP possui escolhas condicionais baseadas na avaliação de expressões lógicas. As expressões podem incluir a avaliação de valores de parâmetros de processo, como também de variáveis cujos valores são definidos a partir de sincronização. Se o resultado da avaliação é verdadeiro (true) a escolha segue por um comportamento, caso contrário, quando falsa (false), a escolha segue o outro fluxo especificado. O construtor condicional básico pode ser visto na Tabela 3.1. Esse processo comporta-se como P se a expressão lógica g for verdadeira, ou como Q, se g for falsa. Como exemplo, a seguir temos o processo *COND*:

$$
\begin{aligned}
&COND(ev, ia) = \\
&\quad if(ev \in ia)\, then \\
&\qquad ev \rightarrow Skip \\
&\quad else \\
&\qquad ev \rightarrow Stop
\end{aligned}
$$

O processo *COND* possui dois parâmetros, *ev* um evento e *ia* um conjunto de eventos. Caso o valor passado para *ev* pertença ao conjunto *ia*, a expressão lógica ($ev \in ia$) será verdadeira, e o processo terminará com sucesso (*Skip*), no caso contrário, quando a mesma expressão for falsa, o processo se comportará como deadlock (*Stop*).

Em CSP, a construção condicional pode ser abreviada para a forma $g \& P$, que equivale a expressão $if\,(g)\,then\,P\,else\,Stop$. O exemplo anterior pode ser abreviado como $(ev \in ia) \& Skip$.

3.1.6 Internalização de Eventos

Quando se quer esconder eventos da especificação tornando-os ações internas, emprega-se o operador de internalização (*hiding*) de CSP. Isso pode ser bastante útil quando se quer impedir que outros processos e o ambiente influenciem no comportamento dos eventos. Depois de escondidos os eventos se tornam invisíveis e incontroláveis pelo ambiente, pois nenhuma sincronização pode ser realizada. Apesar de continuarem existindo e ocorrendo no processo, o ambiente externo não pode vê-los. Esse operador recebe como parâmetros um processo P e o conjunto de eventos C que se quer esconder, como pode ser visto na Figura 3.1. Depois de ocultados os eventos em C, o processo comporta-se exatamente como o processo P, exceto que os eventos que pertencem a C não podem ser visualizados externamente.

Como exemplo, considere o processo $P2$ apresentado na Seção 3.1.3. Escondendo de $P2$ o conjunto de eventos $\{c\}$ temos: $P2 \setminus \{c\}$. O comportamento após a internalização será: oferecer o evento y para sincronização com o ambiente, após a sincronização, o evento c ocorre internamente, em seguida, volta a se comportar como $P2 \setminus \{c\}$.

Através do operador de *hiding* é possível construir um processo cujo comportamento é equivalente ao *DIV* mencionado na Seção 3.1.1. Para isto, basta criar um processo recursivo que tenha todo o seu alfabeto escondido. Como exemplo, temos o processo $P3$ e Q:

$$
\begin{aligned}
&P3 = a \rightarrow P3 \\
&Q = P3 \setminus \{a\}
\end{aligned}
$$

O processo Q se comporta como $P3$ quando todo o seu alfabeto $\Sigma_{P3} = \{a\}$ é escondido: funciona com a recursão infinita $P = P$, que será interpretada pelo ambiente externo como um *livelock*, ou divergência.

3.1.7 Composição Seqüencial

O operador de composição seqüencial (denotado pelo símbolo ;) permite que dois processos sejam executados de acordo com a ordem na qual são compostos pelo operador. É um operador binário que recebe dois processos, de acordo com a Tabela 3.1. O comportamento da composição consiste na "execução"do primeiro processo (que fica antes da símbolo ;) que, quando termina com sucesso, se comporta como o segundo processo, caso o primeiro não termine com sucesso (não tenha terminação ou termine com Stop), o segundo processo nunca será executado.

Por exemplo, o processo

$$Q \rightarrow Skip\,;\ R$$

se comporta inicialmente como o processo Q. Após o término com sucesso de Q (identificado quando este passa a comportar-se como Skip) o processo Q; R passa a comportar-se como R.

3.1.8 Interrupção

Similar ao operador de composição seqüencial, o operador de interrupção é útil para impor uma ordem de prioridade entre dois processos. O processo

$$Q \ \triangle\ R$$

comporta-se como Q até que o processo R o interrompa, a partir de onde o processo comporta-se como R. A interrupção ocorre quando o ambiente sincroniza com qualquer evento que R ofereça.

Como exemplo do operador, temos o processo

$$RUN(\{a,b\}) \ \triangle\ c \rightarrow Skip$$

que se comporta como $RUN(\{a,b\})$ até que $c \rightarrow Skip$ o interrompa. $RUN(\{a,b\})$ oferece a escolha externa entre os eventos a e b enquanto o evento c não é comunicado. Quando o ambiente sincroniza em c, o comportamento de $RUN(\{a,b\})$ é interrompido, e o processo se comporta como $Skip$ (termina com sucesso).

Se Q e R oferecem os mesmos eventos ao ambiente, então ocorre uma escolha não-determinística entre eles.

3.1.9 Paralelismo

Até este ponto, os processos se comportavam apenas seqüencialmente: um processo é a continuação de outro (recursão na Seção 3.1.3), ou, o processo só inicia quando o outro anterior

terminou com sucesso (composição seqüencial na Seção 3.1.7). Através do paralelismo é possível executar mais de um processo simultaneamente, possivelmente havendo comunicação entre eles.

De uma forma geral, em todos os paralelismo apresentados que envolvem sincronização de eventos (síncrono, alfabetizado e generalizado), a sincronização ocorre de forma semelhante quando estão envolvidos 2 processos (chamada de sincronismo simples ou *rendezvous*), ou quando estão envolvidos 3 ou mais processos (chamada de sincronismo complexo ou *multi-way rendezvous*). Durante o *rendezvous*, ambos os processos emissores e receptores permanecem bloqueados no ponto de sincronização, até que todos os envolvidos alcancem este ponto em seus fluxos particulares. Só então, a comunicação ocorre e os processos envolvidos continuam seu fluxo, de forma independente, até que alcancem o próximo ponto de sincronização.

A forma mais simples de paralelismo com sincronização de eventos entre os processos envolvidos chama-se paralelismo síncrono. Sejam Σ_P e Σ_Q os alfabetos de P e Q, e P está em paralelo com Q via paralelismo síncrono

$$P \parallel Q$$

então, cada um dos eventos oferecidos por P aguarda a sincronização com um evento oferecidos por Q, e vice-versa. O processo $P \parallel Q$ só comunica eventos caso P e Q possam oferecer e sincronizar os mesmo eventos. Como conseqüência, os processos comunicam evento a evento em parceria, caso contrário não comunicam. Na situação em que a interseção dos alfabetos é vazia ($\Sigma_P \cap \Sigma_Q = \emptyset$), esse paralelismo se comporta como *Stop*.

A regra (3.1) deixa explícito o comportamento do paralelismo síncrono de P e Q, onde os processos só progridem se sincronizam no evento e oferecido por ambos, feita a sincronização, os processos comunicam o evento e e seguem seus fluxos de comportamento como P' e Q'.

$$\frac{P \xrightarrow{e} P', Q \xrightarrow{e} Q'}{P \parallel Q \xrightarrow{e} P' \parallel Q'} \tag{3.1}$$

Na situação onde ocorre um evento interno τ em algum dos processos, apenas o processo cuja ação interna foi disparada progride, o outro permanece no mesmo ponto, inalterado. Isso pode ser visto nas regras (3.2) e (3.3). Na primeira regra, P comunica uma ação interna e progride (avança no seu fluxo se eventos) sozinho tornando-se P'. Na segunda regra, Q comunica uma ação interna e progride sozinho tornando-se Q'.

$$\frac{P \xrightarrow{\tau} P'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q} \tag{3.2}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \parallel Q \xrightarrow{\tau} P \parallel Q'} \tag{3.3}$$

De forma diferente, sejam X um conjunto tal que $X \subseteq \Sigma_P$, e Y um conjunto tal que $Y \subseteq \Sigma_Q$, o paralelismo alfabetizado dos processos P e Q anteriores, é denotado pelo processo

$$P \,{}_X\!\parallel_Y\, Q$$

que executa os processos P e Q sincronamente em todos os eventos do conjunto $X \cap Y$. Se um dos eventos dessa interseção for aceito por apenas um dos processos, ele não poderá ocorrer em qualquer dos dois, pois é obrigatória a sincronização. Só os eventos fora da interseção dos alfabetos podem ser comunicados independentemente.

Uma outra variação do operador de paralelismo, chamada *interleaving*, permite compor processos em paralelo, sem que haja interações entre eles. Cada evento oferecido ao *interleaving* de dois processos ocorre apenas em um deles. Caso ambos estejam dispostos a aceitar um mesmo evento, a escolha entre os processos é não-determinística. Esse tipo de combinação é especificado da seguinte forma:

$$P \,|||\, Q$$

O comportamento do *interleaving* de dois processos é idêntico ao paralelismo alfabetizado de dois processos, quando a interseção dos alfabetos é um conjunto vazio.

Os três operadores de paralelismo apresentados acima podem ser representados por um único operador, chamado paralelismo generalizado. Através deste operador, basta informar o conjunto de sincronização que contém os eventos que devem ocorrer simultaneamente (sincronizados) entre os processos participantes, eventos fora deste conjunto são executados como no *interleaving*, de forma independente. Abaixo, está a representação dos operadores de paralelismo síncrono, paralelismo alfabetizado e *interleaving* através da notação do operador de paralelismo generalizado.

$$\begin{array}{lcl} P \parallel Q & \equiv & P |[\, \Sigma_P \cup \Sigma_Q \,]| Q \\ P_X \parallel_Y Q & \equiv & P |[\, X \cap Y \,]| Q \\ P \,|||\, Q & \equiv & P |[\, \{\} \,]| Q \end{array}$$

3.2 Semântica denotacional vs operacional

A representação mais comum e poderosa de CSP chama-se semântica denotacional: representação algébrica do comportamento dos processos, que é o formato que temos visto neste capítulo até este ponto. Esta representação é toda baseadas em leis algébricas permitem a verificação com rigor matemático de propriedades clássicas de sistemas concorrentes e distribuídos, como o determinismo e a ausência de *deadlock* ou *livelock*. Abaixo exemplificamos duas destas leis [RHB97]:

$$P \,\Box\, Stop \equiv P$$
$$P \,|||\, Skip \equiv P$$

As leis algébricas de CSP permitem provar equivalências semânticas (através de refinamentos) entre processos sintaticamente diferentes. Algumas destas leis podem ser usadas para reescrever a definição de processos, tornado-os mais simples ou atendendo algum padrão estrutural, sem no entanto mudar o comportamento do processo original.

Outra representação chamada semântica operacional é utilizada como um modelo que é manipulado por ferramentas (ex: FDR, [Sys05]) que implementam a verificação de propriedades e refinamentos entre processos (descritas na Seção 3.3). A semântica operacional de CSP é todas descrita a partir de sistemas de transições rotuladas (LTS).

Formalmente, um LTS é uma quádrupla $M = (Q, A, \rightarrow, q_0)$, Q é um conjunto finito não vazio de estados, $q_0 \in Q$ é o estado inicial, A é o alfabeto de ações e $\rightarrow$ é a relação de transição tal que $\rightarrow \subseteq Q \times A \times Q$. É um grafo direcionado, cujas transições estão rotuladas pelos eventos que pertencem ao conjunto de ações A e seguem a relação de transição $\rightarrow$.

Todo processo P finito, pode ser representado por uma LTS, de forma que o conjunto de ações do LTS é igual ao alfabeto de P (Σ_P). Para os construtores básicos de processos apresentados acima na Seção 3.1 (com excessão do operador de paralelismo) existe um mapeamento direto de CSP para sua semântica operacional representada em LTS.

A Figura 3.1 corresponde ao LTS que modela o comportamento do processo *Stop*. Este processo primitivo é representado por um estado terminal do grafo que não possui transições de saída a partir dele.

Figura 3.1 Semântica Operacional do processo STOP

A Figura 3.2 corresponde ao LTS que modela o comportamento do processo *Skip*. Este processo primitivo é representado por um estado inicial (na Figura rotulado) e uma transição de saída rotulada por ✓ que leva ao estado terminal do grafo.

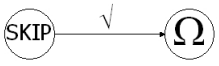

Figura 3.2 Semântica Operacional do processo SKIP

A Figura 3.3 corresponde ao LTS que modela o comportamento do processo $a \rightarrow P$ que utiliza o operador de prefixo. Tal processo é representado por um estado inicial (na Figura sem rótulo) e uma transição de saída rotulada pelo evento de prefixo a, que leva a um estado não terminal do grafo (na Figura rotulado como P). Este estado não terminal corresponde ao estado inicial do LTS obtido a partir da representação operacional do processo P.

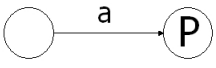

Figura 3.3 Semântica Operacional do operador de prefixo

A Figura 3.4 corresponde ao LTS que modela o comportamento do processo recursivo $P = a \rightarrow P$. Tal processo é representado por um único estado (na Figura rotulado) e uma transição rotulada pelo evento a, do estado para ele próprio (loop).

A Figura 3.5 corresponde ao LTS que modela o comportamento do operador de escolha externa do processo $a \rightarrow P \,\square\, b \rightarrow Q$. Tal processo é representado um estado inicial (na Figura

Figura 3.4 Semântica Operacional de um processo recursivo

não rotulado) e duas transições de saída. Uma transição rotulada pelo evento *a* que leva a um estado não terminal (na Figura rotulado por *P*). Este último estado representa o estado inicial do LTS que modela o comportamento do processo *P*. E outra transição rotulada pelo evento *b* que leva a um estado não terminal (na Figura rotulado por *Q*). Este último estado representa o estado inicial do LTS que modela o comportamento do processo *Q*.

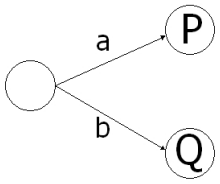

Figura 3.5 Semântica Operacional do operador de escolha externa

A Figura 3.6 corresponde ao LTS que modela o comportamento do operador de escolha interna do processo $a \rightarrow P \sqcap b \rightarrow Q$. Tal processo é representado um estado inicial (na Figura não rotulado) e duas transições de saída. Uma transição rotulada pelo evento τ (representa uma ação interna do processo) que leva a um estado não terminal (na Figura rotulado por *P*). Este último estado representa o estado inicial do LTS que modela o comportamento do processo *P*. E outra transição rotulada pelo mesmo evento τ que leva a um estado não terminal (na Figura rotulado por *Q*). Este último estado representa o estado inicial do LTS que modela o comportamento do processo *Q*.

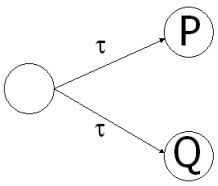

Figura 3.6 Semântica Operacional do operador de escolha interna

A intuição da representação operacional de processos é importante para entender os LTSs mostrados na Seção.5.

3.3 Prova de propriedades e Refinamentos

CSP usa modelos semânticos para estabelecer relações de refinamento entre processos, a partir de onde as leis podem ser aplicadas. Os principais modelos semânticos são *Traces*, *Falhas* e *Falhas e Divergências*. Relações de refinamento formalizam a idéia de implementação correta com respeito à especificação: se uma implementação é um refinamento de uma especificação, então é possível provar que a implementação satisfaz as propriedades (requisitos) descritas na especificação.

O modelo de *Traces* é útil para encontrar sequências finitas de eventos que um processo pode executar, e verificar até que ponto um padrão comportamental é atendido. Um trace de um processo é um registro de uma sequência finita de eventos que o processo já aceitou até o momento. Por exemplo:

- $\langle\rangle$ - é o trace de um processo que ainda não aceitou evento algum. A semântica denotacional de CSP exige que todo processo possua este trace.
- $\langle x, y\rangle$ - é um trace de um processo que aceita o evento x, seguido do evento y;
- $\langle\checkmark\rangle$ - é o trace do processo Skip;
- $\langle x\rangle$ - é o trace de um processo que aceita o evento x e em seguida realiza uma ação internalizada. Ações internas não são capturadas pelo modelo de traces.

Entretanto, o modelo de *Traces* não é capaz de identificar os eventos que um processo não pode executar num determinado instante. Assim, segundo este modelo, o processo $P \,\Box\, Q$ é equivalente ao processo $P \sqcap Q$, visto que ambos podem, eventualmente, ter o mesmo comportamento. Exemplificando, seja o processo P equivalente $a \rightarrow Skip \,\Box\, b \rightarrow Stop$, e o processo Q equivalente a $a \rightarrow Skip \sqcap b \rightarrow Stop$, então:

$$traces(P) = traces(Q) = \{\langle\rangle, \langle a\rangle, \langle a, \checkmark\rangle, \langle b\rangle\}$$

As observações consideradas para responder se um processo de implementação I, é um refinamento válido (no modelo de traces) do processo de especificação S, é exatamente o conjunto de traces da implementação e da especificação. Uma implementação I refina a especificação S pelo modelo de traces se:

$$traces(I) \subseteq traces(S)$$

Uma vez calculado que os traces produzidos pela implementação I, são iguais ou menores que os traces produzidos pela especificação S, então, o refinamento no modelo de traces é válido e denotado por:

$$S \sqsubseteq_{\tau} I$$

Lê-se, que o processo I refina o processo S pelo modelo de traces.

O modelo de *Falhas* é mais poderoso que o de *Traces* porque permite identificar a sequência de eventos que um processo pode realizar, juntamente com o conjunto de eventos que ele pode

recusar-se a executar naquele momento. O modelo de *Falhas* inclue o de *Traces*. Assim, este modelo permite demonstrar que o processo $P \Box Q$ não é semânticamente equivalente ao processo $P \sqcap Q$, se considerarmos o conjunto de eventos que podem não ser aceitos. Através do modelo de *Falhas* também é possível verificar se um processo é determinístico ou não. Um processo é dito determinístico se ele não se comporta diferentemente a partir da mesma situação inicial.

O modelo de *Falhas e Divergências* é ainda mais poderoso que o de *Falhas*, e consequentemente mais poderoso que o de *Traces*: inclui os dois anteriores. Este modelo identifica todas os traces que podem levar um processo a comportar-se como uma divergência (*DIV*), ou *livelock*. Por ser o modelo mais forte e completo de CSP, é utilizado para provar equivalência de comportamento entre processos. Dois processos são equivalentes, isto é, possuem comportamento idêntico, se ambos possuem o mesmo modelo de *Falhas e Divergências*. Se P é equivalente a Q, então:

$$P \sqsubseteq_{\mathcal{FD}} Q$$
$$Q \sqsubseteq_{\mathcal{FD}} P$$

A primeira expressão lê-se, Q refina pelo modelo de *Falhas e Divergências* o processo P, isto é, Q possui uma quantidade de traces, falhas e divergências menor ou igual ao processo refinado (P). A segunda expressão lê-se, P refina pelo modelo de *Falhas e Divergências* o processo Q, isto é, P possui uma quantidade de traces, falhas e divergências menor ou igual ao processo refinado (Q). Ambos os refinamento só são válidas se os processos P e Q são equivalente: possuem exatamente a mesma quantidade de traces, falhas e divergências.

O foco desse trabalho são os cálculos de refinamentos no modelo de *Traces*.

3.4 Suporte de Ferramentas

A ferramenta mais conhecida para prova de refinamentos sobre processos CSP é o FDR (Failures and Divergences Refinament) [Sys05]. O ProBE [Sys03] é outra ferramenta útil para animar modelos CSP. Ambas lêem especificações CSP descritas em uma linguagem funcional chamada CSP_M [Sys05], que é um acrônimo para *machine readable CSP*.

A tabela 3.2 mostra na notação CSP_M, a relação dos operadores de processo listados na tabela 3.1.

Em acréscimo às definições da tabela 3.2, temos outras construções CSP_M que serão empregadas:

- P`|||`P — Entrelaçamento entre dois processos;
- `[]`x`:`a`@`p — Escolha externa indexada;
- $s1$`^`$s2$, `head`(s) e `tail`(s) — Correspondem, respectivamente, aos operadores de seqüência: concatenação de listas (concatena a listas $s1$ com a lista $s2$), retorna a cabeça da lista s e retorna a calda da lista s; e,

$P(s)$	::=	`STOP`	
	|	`SKIP`	
	|	a `->` P	(prefixo)
	|	$P(s)$	(recursão)
	|	$P(s)$ `[]` $P(s)$	(escolha externa)
	|	$P(s)$ `|~|` $P(s)$	(escolha interna)
	|	`if` (g) `then` $P(s)$ `else` $P(s)$	(escolha condicional)
	|	$P(s) \setminus C$	(internalização)
	|	$P(s)$ `;` $P(s)$	(composição seqüencial)
	|	$P(s) \,/\backslash\, P(s)$	(interrupção)
	|	$P(s)$ `[|`C`|]` $P(s)$	(paralelismo)

Tabela 3.2 Operadores de CSP_M

- `union`$(C1, C2)$, diff$(C1, C2)$ e member(e, C) — Correspondem, respectivamente, aos operadores de conjunto: união de conjuntos (retorna a união dos conjuntos $C1$ e $C2$), diferença entre conjuntos (retorna a diferença dos conjuntos $C1$ e $C2$) e teste de presença de elemento (retorna verdade o elemento e pertence ao conjunto C e falso no caso contrário).

Capítulo 4

Testes Formais

Métodos formais e teste de software possuem essecialmente objetivos similares no desenvolvimento do software: produzir software de alta qualidade, minimizando e até evitando a presença de faltas nos diversos artefatos de desenvolvimento. Já foram (e ainda são) vistas em algumas comunidades como sendo completamente independentes, não tendo qualquer relacionamento entre si. Entretanto, uma combinação entre ambas vem emergindo como uma nova linha de pesquisa bem interessante denominada de testes formais [BBC+03] [BBC+02].

Enquanto métodos formais objetivam documentar precisa e concisamente a especificação de um sistema e usá-la para provar propriedades acerca deste sistema, teste de software executa a implementação do sistema para constatar a presença ou não de falhas, tomando como base a especificação do sistema. Assim sendo, testes formais são testes que tomam como base artefatos formais (modelos) tanto da especificação do sistema quanto da implementação. Tais modelos podem ser processados por ferramentas que automatizam várias atividades do ciclo de testes. Testes formais permitem reduzir o custo de implementação pela aplicação de técnicas de teste com maior antecedência no ciclo de vida, enquanto defeitos são relativamente mais baratos de corrigir, e aumentar o nível de automação do processo de teste pela:

- Geração de casos de teste funcionais a partir da especificação do sistema; e,
- Derivar oráculos para verificar os resultados dos testes.

A Figura 4.1 reusa a Figura 2.1 para mostrar onde é possível fazer uso de métodos formais e testes formais. Os focos principais são:

- Propriedades da especificação podem ser provadas através de provas de teoremas (2), quando o modelo for infinito, e verificação de modelos (1), quando o modelo for finito;
- A especificação pode ser validada através de sua animação (3);
- Critérios de seleção bem definidos podem ser aplicados ao modelo referente à especificação (4), por exemplo, cobertura de todas as condições lógicas, ou cobertura de todos os caminhos através de uma máquina de estados;
- Casos de teste funcionais, em nível de sistema, podem ser gerados a partir das especificações (5);
- Um oráculo pode ser derivado para verificar os resultados das execuções de teste de forma automática (6);

- Gerenciamento de teste pode ser aprimorado, raciocinando sobre a seqüência de testes (7).
- Projeto e o código podem ser verificados contra a especificação (8 e 9);
- Métodos formais podem sugerir novos tipos de modelos de execução do código (10), por exemplo, a implementação se representado como uma máquina de estados; e,
- Novos modelos de execução permitem: verificar (11) e provar propriedades deste modelo (12), e utilizar critérios de adequação de teste específicos para o modelo (13).

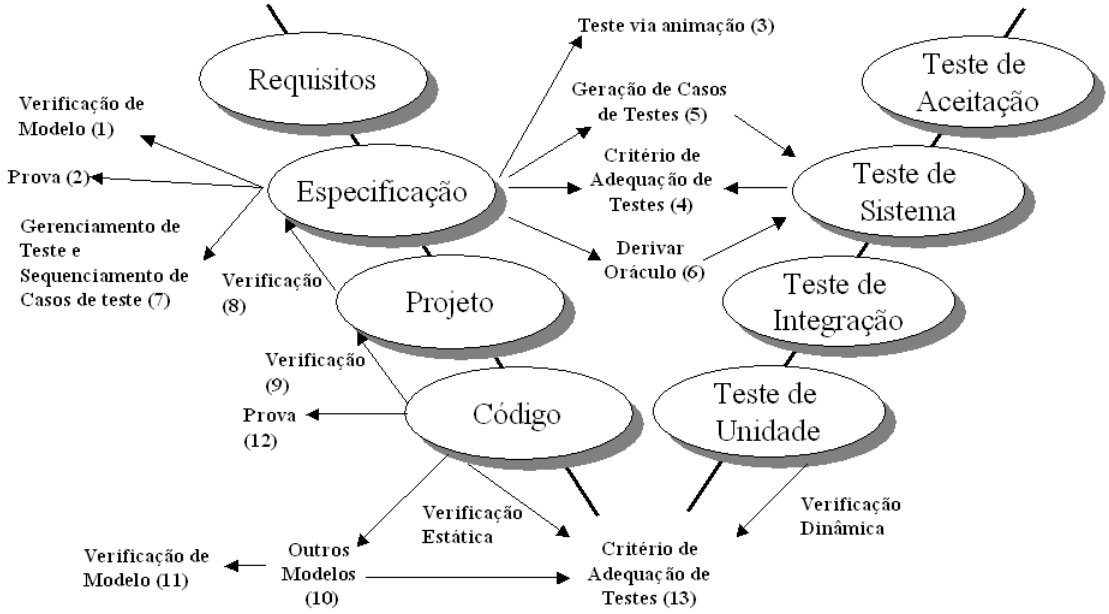

Figura 4.1 Possibilidades de utilização dos métodos formais com testes

4.1 Fundamentos de Testes Formais

Para entender o relacionamento entre teste e métodos formais, é preciso entender o relacionamento entre teste e prova. Nesta seção são descritos alguns fundamentos baseados nos trabalhos [Gau95] e [BBC+03].

A estrutura na Figura 4.2 propõe duas situações limite:

- Em (1), extremo superior esquerdo, onde nenhuma prova é necessária mas uma quantidade infinita de testes é necessária para verificar o sistema, e;

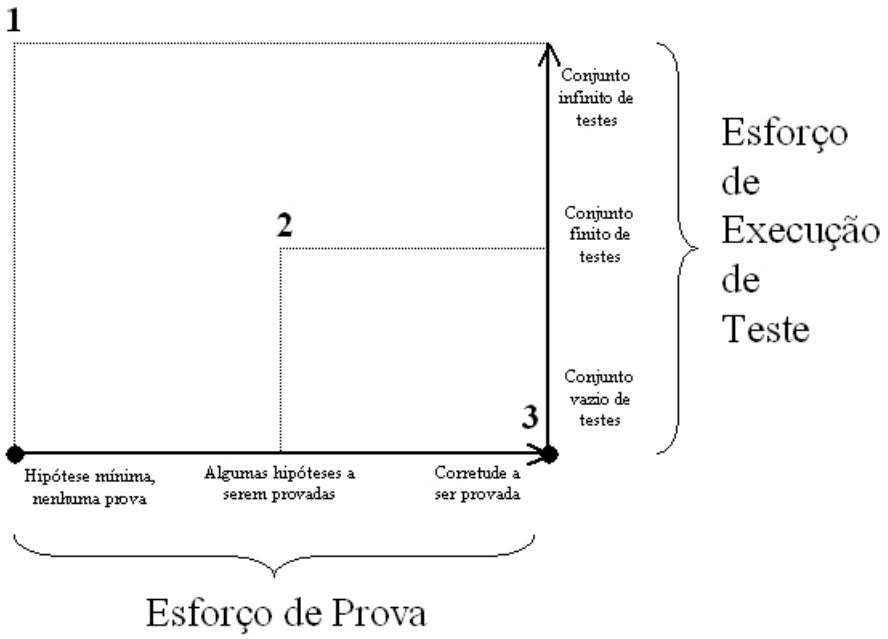

Figura 4.2 Estrutura para relacionamento entre prova e teste

- Em (3), extremo inferior direito, nenhum teste é necessário porém, uma prova completa de corretude do programa com relação a especificação é requerida.

Em (2) podemos ver um exemplo de situação intermediária que pode ser encontrada movendo-se entre os dois limites apresentados anteriormente. Em tal situação, algumas hipóteses precisam ser provadas com relação à IUT e uma quantidade finita de testes precisa ser executada contra a implementação. A seguir, ilustramos alguns exemplos de tais hipóteses:

- Se o programa estiver correto (reproduz o resultado esperado) para um conjunto de valores de entrada dentro de um domínio, então ele também deve estar correto para todos os outros valores de entrada do mesmo domínio;

- Seja k uma constante natural. Se o programa estiver correto para alguns valores n onde $n \leq k$, então ele também funcionará corretamente para todos os possíveis valores $n > k$. Por exemplo, se o programa funciona para os valores 0, 1 e 2 ($k = 2$), conseqüentemente, funcionará com os valores $n > 3$; e,

Tais hipóteses são chamadas de hipóteses de seleção de teste. A primeira e a segunda foram definidas em [Gau95], respectivamente como, hipótese da uniformidade (*uniformityhypothesis*) e hipótese de regularidade (*regularityhypothesis*).

Declarando a hipótese de seleção de testes e o conjunto de casos de teste resultante, a tarefa de validação remanescente consiste em demonstrar a execução bem sucedida dos testes, e provar a hipótese para o conjunto de testes escolhido. Retornando aos extremos (1) e (3) da Figura 4.2, pode-se inferir que é inviável executar infinitos testes (1), da mesma maneira que

provar a corretude do programa completo (3) também o é. Mesmo na situação intermediária (2), ainda existe um grande esforço de provar a hipótese para cada teste do conjunto finito selecionado para execução, o que torna o trabalho impraticável. Com isso, fica mais claro que, em um contexto prático industrial, tanto teste quanto prova podem aumentar o nível de confiança com respeito à corretude de um sistema porém, nunca demonstrar isso absolutamente.

4.1.1 Estrutura formal para teste

A estrutura (framework) teórica para teste com suporte dos métodos formais que será apresentada a seguir está presente em [Gau95].

Considerando-se um programa P como uma função parcial de valores concretos de entrada I para valores concretos de saída O então temos que:

$$P \in I \nrightarrow O \tag{4.1}$$

É assumido que P termina a execução para todos os valores de entrada do seu domínio, $dom(P)$. O termo $P(t)$ representa o resultado de executar P com o valor de entrada $t \in dom(P)$. O valor t pode ser visto como um dado de teste fornecido ao programa P.

Uma especificação S é a relação entre valores abstratos de entrada, D, e de saída, R:

$$S \in D \leftrightarrow R \tag{4.2}$$

A definição (4.2), por ser relacional, permite com que a especificação aceite diversos valores de saída para um mesmo valor de entrada. Para um dado $d \in dom(P)$, o conjunto de resultados da especificação é a imagem do conjunto $\{d\}$ em S, e é dado por $S(|\ d\ |)$. Para comparar os resultados de um programa (4.1) contra sua especificação (4.2), deve existir uma relação para mapear valores abstratos em valores concretos. As seguintes funções de abstração são utilizadas:

$$A_D \in I \nrightarrow\!\!\!\!\rightarrow D \tag{4.3}$$

$$A_R \in O \nrightarrow\!\!\!\!\rightarrow R \tag{4.4}$$

Como todo valor abstrato necessita ter pelo menos uma representação concreta, as funções (4.3) e (4.4) são definidas como sobrejetivas. As duas funções mapeiam valores concretos em abstratos, a primeira entradas concretas em saídas abstratas, a segunda saídas concretas em saídas abstratas. Como resultado, todo elemento de D possui um mapeamento no domínio de A_D e, todo elemento de R possui um mapeamento no domínio de A_R. A existência de tais funções, quando se aplicam e existem no contexto, é denominada hipótese mínima, denotada como H_{MIN}.

Falando sobre corretude de P com relação à S, a exigência é que P exiba o comportamento especificado em S. Para isso, primeiramente, todos os valores abstratos de entrada em $dom(S)$ precisam ter uma representação concreta para que P use como entrada. Formalmente, temos que a imagem da inversa de (4.3) aplicada a $dom(S)$ deve estar contida em $dom(P)$.

$$A_D^{-1}(|\ dom(S)\ |) \subseteq dom(P) \tag{4.5}$$

Segundo, o resultado de executar o programa em qualquer teste $t \in dom(P)$ deve estar correto com respeito à especificação. Lembrando que, algum comportamento de P pode estar fora do escopo da especificação, nesse caso, qualquer comportamento é aceito.

$$\forall\, t \in dom(P) \bullet A_D(t) \in dom(S) \Rightarrow A_R(P(t)) \in S(|\ \{A_D(t)\}\ |) \tag{4.6}$$

Simplificando, a definição (4.6) é valida para um dado P, quando a execução de $P(t)$ for correta com relação à S ou simplesmente quando $CORRECT(P,S,t)$ for válido. Esta definição pode ser estendida para o caso da execução de um conjunto de testes T.

$$CORRECT(P,S,T) \Leftrightarrow \forall\, t \in T \bullet CORRECT(P,S,t) \tag{4.7}$$

Resolver (4.6) é conhecido como o problema do Oráculo. Este problema consiste em definir se o resultado de uma execução de teste está ou não dentro do comportamento esperado. Essa estrutura equivale ao diagrama de comutação da Figura 4.3.

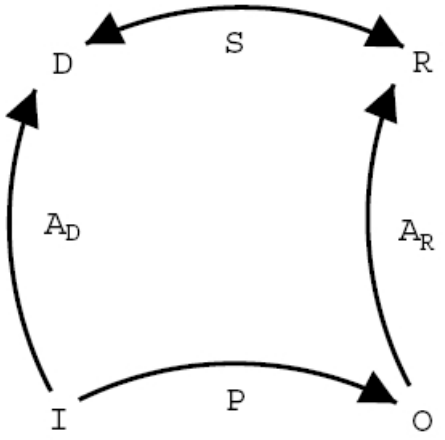

Figura 4.3 Corretude de um programa com respeito a especificação

A idéia associada ao diagrama da Figura 4.3 é que a composição da função P e A_R esteja contida nas relações resultantes da composição de A_D e S:

$$P;\ A_R \subseteq A_D;\ S \tag{4.8}$$

A prova completa da corretude do programa equivale a demonstrar a relação (4.8). Teste exaustivo consiste em executar o programa em toda entrada possível, potencialmente um conjunto infinito, e verificar cada resultado. Esse teste exaustivo é chamado T_{MAX} e deve conter pelo menos as entradas de $A_D^{-1}(dom(S))$.

Usando os últimos conceitos nos elementos essenciais da Figura 4.2 temos que, o programa P está correto com respeito à especificação S se:

- Em (1), a hipótese mínima H_{MIN} pode ser provada e, todos os testes do conjunto T_{MAX} passam;
- Em (3), $H_{MIN} \wedge P$; $A_R \subseteq A_D$; S pode ser provado. Não é necessário executar testes; e,
- Em (2), há diferentes opções para H e T, que correspondem a provar hipóteses de seleções de testes e executar os testes selecionados.

A estrutura apresentada na Figura 4.2 serve para enfatizar a importância de definir, para todo conjunto de testes selecionado T, o conjunto de hipóteses H que expressa a adequação ao conjunto T. Aplicando os elementos deste estrutura, podemos formalizar as hipóteses de uniformidade e regularidade apresentadas anteriormente.

Uniformidade Assume que existe uma homogeneidade sobre uma faixa de dados de teste. Se um teste é correto em um conjunto uniforme T, a hipótese é que todo teste em T esteja correto. Dados P, S e T então: $\forall t \in T \bullet CORRECT(P,S,t) \Rightarrow CORRECT(P,S,T)$.

Regularidade Assume que existe uma regularidade sobre uma estrutura, ou tamanho, dos dados. Nesse caso, se os dados dos testes estão corretos até um certo limite de tamanho *tam*, a hipótese é que todo teste que utilize um valor maior que este limite também estará correto. Dados P, S e T, e uma função de tamanho *size*, $T' = \{t \in T \bullet size(t) < tam\}$ em $CORRECT(P,S,T') \Rightarrow CORRECT(P,S,T)$.

Em [Gau95] também é introduzida a idéia de mover para cima e para baixo o espectro de seleção de testes por meio de um refinamento. Considere dois pares hipóteses/testes (H_1,T_1) e (H_2,T_2). Então (H_1,T_1) refina (H_2,T_2) se:

- A hipótese se torna mais forte, isto é, $H_1 \Rightarrow H_2$, e;
- A habilidade de T_1 detectar defeitos é pelo menos tão forte quanto T_2.

Formalmente, (H_1,T_1) refina (H_2,T_2) se:

$$H_1 \Rightarrow H_2 \wedge (H_1 \wedge CORRECT(P,S,T_1) \Rightarrow CORRECT(P,S,T_2)) \tag{4.9}$$

Usando esta idéia, a estrutura sugere uma abordagem incremental para o desenvolvimento do conjunto de testes na qual, um conjunto infinito de testes é gradualmente refinado para um conjunto viável de trabalho (conjunto finito), enquanto o correspondente conjunto de hipóteses de seleção é coletado. Considerando a hipótese mínima H_{MIN}, o primeiro conjunto de testes selecionados é infinito, a medida que surgem outras hipóteses H_2, H_3 e H_N, onde as posteriores são consecutivamente mais fortes que as anteriores, o conjunto de teste é paulatinamente refinado para outros com uma quantidade mais restrita de testes.

4.1.2 Critérios de Seleção

Na Seção 4.1.1 é fornecida uma estrutura para testes formais como também uma técnica de seleção de testes baseadas em hipóteses. De forma mais ampla e resumida, de acordo com [Gau04], podemos classificar as principais abordagens de seleção de um subconjunto de testes como:

1. Critério de Cobertura;
2. Hipótese de seleção; e,
3. Propósito de Teste.

Os dois primeiros critérios são implícitos, isto é, estão congelados dentro do algoritmo de geração, não podendo ser alterados; o terceiro é um critério explícito, pois o critério de seleção é variável e fornecido pelo usuário como entrada para a abordagem [FHP02].

Seleção via critério de cobertura é baseada no modelo da especificação, como conseqüência, varia de acordo com o tipo do modelo. Pode-se enumerar alguns critérios de cobertura bastante comuns: cobertura de requisitos (*requirements coverage*), cobertura de estados (*state coverage*), cobertura de comandos (*statement coverage*), cobertura de condicionais (*branch coverage*), cobertura de faltas (*fault coverage*), etc. No caso das máquinas de estado finitas é muito conhecido a cobertura de transições [ben99]. Um conjunto de testes derivados a partir de tal cobertura explora todas as possíveis transições da máquina de estados.

Seleção via hipótese de seleção é uma abordagem baseada na idéia de fortalecer a hipótese de teste. Recordando da Seção 4.1, hipótese de teste são restrições e afirmativas que assumimos como verdade para a classe de IUT, por exemplo, a IUT deve funcionar de forma semelhante ao modelo, ou, as ações do IUT devem ser atômicas assim como no modelo para garantir um comportamento coerente com o modelo. Hipóteses de teste são importantes para prevenir comportamentos não-determinísticos (demoníacos) da implementação, isto é, podem se comportar bem durante os testes mas incorretamente quando executados no ambiente de produção. LOFT [Mar95] é uma ferramenta de geração baseada na hipótese de uniformidade [Gau95] para gerar testes a partir de especificações algébricas.

Quando o objetivo do teste é focar na verificação de propriedades ou comportamentos importantes do sistema a ser testado, ao invés de provar equivalência de comportamento entre especificação e implementação [GHNS93], podemos utilizar propósitos de teste (TP — *test purposes*), que definem as propriedades a serem verificadas. Esta abordagem será descrita em mais detalhes na subseção seguinte, já que é utilizada na técnica de geração de testes proposta por este trabalho no Capítulo 5.

4.1.2.1 Propósitos de Teste

Na seleção por propósitos de teste as hipóteses assumidas sobre a IUT são bem mais simples (com relação a seleção via hipóteses de seleção) como, assumir que a implementação pode ser modelado como um grafo, ou, que a implementação possui um alfabeto bem definido e

conhecido. Um propósito de teste[1] pode descrever fluxo de controle, fluxo de sinais, fluxo de dados, tempo e probabilidade para especificar propriedades que se desejam testar. Sua definição é um processo intuitivo e usualmente realizado por humanos. Os testes gerados representam a intersecção entre o modelo e os propósitos de testes fornecidos.

Propósitos são normalmente descritos usando um dos seguintes formatos:

MSCs Em [SEG00], MSC é utilizada para especificar propósitos de teste. Message Sequence Charts (MSC) [Soc06] é uma linguagem gráfica e textual para descrição e especificação de interações entre componentes de sistema. Pode ser usado para especificar requisitos, simular e validar, especificar casos de teste e documentar sistemas de tempo real. Através de propósitos de teste MSC, um desenvolvedor de testes pode optar por escolher certos aspectos do sistema para que sejam cobertos durante a geração de testes, como exemplo, transições, processos ou blocos de especificação.

IOLTSs Em [JJ05], um propósito de testes é modelado como um IOLTS *TP*. IOLTS (*Input-Output LTS*) é uma 4-tupla *TP*

$$TP = (Q^{TP}, A^{TP}, \longrightarrow^{TP}, q_0^{TP})$$

onde Q^{TP} é um conjunto finito de estados, A^{TP} é um conjunto finito de eventos, $\longrightarrow^{TP}$ é uma relação de transição (que é subconjunto de $\mathbb{P}(Q^{TP} \times A^{TP} \times Q^{TP})$), e q_0^{TP} é um conjunto de estados iniciais. *TP* é utilizado para selecionar comportamentos de uma especificação IOLTS, classificando-os como aceitos (*accepted*) ou recusados (*refused*), para tanto, é equipado com dois conjuntos de estados de armadilha (*trap states*): $Accept^{TP}$ e $Refuse^{TP}$. Possui o mesmo alfabeto de eventos da especificação. É completo e determinístico. Completo porque cada estado do propósito permite que ocorram todas as ações do alfabeto, isto é, para todo estado do propósito de teste ($q \in Q^{TP}$) existe uma transição habilitada para todos os eventos do seu alfabeto: $a \in A^{TP} : q \xrightarrow{a}^{TP}$.

Lógica Temporal Em [Hol04], propósitos de teste são expressos como uma propriedade de lógica temporal. Lógica Temporal [Lam94] suporta a formulação de afirmativas (*assertions*), descritas como fórmulas de lógica temporal, sobre o comportamento do sistema considerando aspectos temporais. Tais fórmulas permitem que projetistas definam restrições sobre aspectos isolados do comportamento do sistema e foquem na especificação de cenários. Esse fato permite que lógica temporal também seja adequada para especificar seqüências de teste.

As fórmulas de Lógica Temporal podem descrever propriedades *safety* ou *liveness* do sistema especificado. Propriedades *safety* definem o que sempre é verdade, como exemplo, um protocolo de comunicação deve sempre estar livre de deadlocks (deadlock-free).

[1] Existem ainda outras definições sobre propósitos de teste, como no contexto das telecomunicações [ETS96], onde propósitos de teste são parte de uma especificação de testes. São definidos em paralelo com o documento de Projeto de Testes com o objetivo de definir, textualmente, os objetivos de cada teste bem como a maneira na qual os testes serão utilizados. Entretanto, neste trabalho, nos reportaremos ao significado formal dos propósitos de teste onde o mesmo é empregado como uma das entradas para as técnicas de seleção de testes.

Propriedades *liveness* refletem o que o sistema em algum momento no futuro (eventualmente) deve satisfazer, como exemplo, sempre que uma mensagem for originada de um aparelho móvel através da rede sem fio para outro aparelho, é garantido que eventualmente a mensagem será entregue no aparelho de destino.

Várias técnicas de geração propostas envolvem uma especificação (parcial) do sistema *S*, e uma propriedade em lógica temporal *P* (propósito de teste) a ser verificada contra *S*. Casos de teste são então gerados percorrendo a especificação com o objetivo de achar seqüências de execução possíveis em *S* que reflitam o propósito de teste *P*.

Nas próximas seções apresentamos, informalmente (mais a título de conhecimento) testes formais associados a determinados formalismos tradicionais.

4.2 Teste Formais Baseados em Modelo

As notações Z [Ire97], B [JR96] e VDM [BJ78] fazem parte de uma categoria de métodos formais denominada baseada em modelos. Permitem a descrição de sistemas em termos de estados e operações abstratas sobre esses estados. Os estados são tipicamente descritos usando modelos matemáticos, tais como conjuntos, seqüências, relações e funções. As operações são descritas por predicados dados em termos de pré e pós-condições. A seguir está um exemplo de um esquema (*scheme*) Z utilizado para especificar estruturas de dados e operações sobre estas.

Stack

$$items : \mathrm{seq}\, Object$$

$$\#items \leq maxsize$$

Assumindo que *Object* é um tipo declarado anteriormente e *maxsize* é uma constantes inteira que define um limite superior de tamanho, o esquema *Stack* especifica a estrutura de uma pilha cuja variável *items* é uma seqüência de objetos ($\mathrm{seq}\, Object$). O invariante (que sempre será verdade em qualquer operações que consulte ou modifique o esquema) dessa pilha é que o tamanho máximo permitido para *items* seja menor ou igual ao valor de *maxsize* ($\#itens \leq maxsize$). O esquema de inicialização a seguir define o estado inicial da pilha.

StackInit

$$Stack'$$

$$items' = \langle\rangle$$

O símbolo $'$ (apóstrofo) é utilizado em variáveis para determinar seu valor após a realização de uma operação (pós-condição). No esquema *StackInit* a pilha ($Stack'$) é inicializada com a seqüência de objetos vazia ($items' = \langle\rangle$), conseqüentemente a pós-condição de *StackInit* é que a pilha esteja vazia. Uma vez inicializada, podemos especificar duas operação sobre esta pilha.

Pop

$\Delta Stack$
$x! : Object$

$items \neq \langle\rangle$
$x! = \text{head}\, items$
$items' = \text{tail}\, items$

Push

$\Delta Stack$
$x? : Object$

$items' = \langle x? \rangle \frown items$

O primeiro esquema *Pop* especifica a operação de remover o elemento no topo da pilha e o segundo esquema *Push* especifica a operação de acrescentar um novo elemento no topo da pilha. É utilizada a convenção $\Delta Stack$ (Δ antes do nome do esquema) em ambos as operações para indicar que elas modificam o estado da pilha (*Stack*). Os símbolos ? e ! utilizados depois do nome da variável x indicam, respectivamente, que um valor do tipo *Object* é fornecido como entrada (na operação *Push*) e um valor do tipo *Object* é retornado como saída (na operação *Pop*). A operação *Pop* tem como pré-condição que a pilha não esteja vazia ($items \neq \langle\rangle$), se esta condição é verdade, é retornado o valor do topo da pilha ($x! = \text{head}\, items$) e este valor é removido ($items' = \text{tail}\, items$). Já a operação *Push*, acrescenta no top da pilha o objeto fornecido como entrada ($items' = \langle x? \rangle \frown items$).

Tomando com referência linguagens de programação orientadas a objetos como Java ou C++, esquemas Z podem ser utilizados para especificar classes e métodos.

Formalismos baseados em modelo provêem uma metodologia rigorosa para verificar e validar as propriedades do sistema e também são úteis para o processo de testes de variadas formas:

- Geração de casos de teste a partir das especificações formais;
- Automação do processo de teste; e,
- Formalização de estratégias de teste, abordagens e estruturas.

Na listagem abaixo estão descritas algumas das técnicas de teste baseadas em modelos formais.

Teste de Partição É uma técnica bastante utilizada com métodos baseados em modelos. Em [V.98], é descrita uma abordagem geral para testar a partir Z pela classificação de domínios de teste. A idéia consiste em obter as várias combinações entre entradas, saídas e estados para obter diferentes cenários de teste. A partir da declaração da especificação, e das condições contidas nos predicados, são obtidas partições dos conjuntos de entrada, saída e dos estados em subconjuntos (sub-domínios) que são combinados. Em [DF93],

demonstra-se que pela reescrita de especificações VDM para a forma disjuntiva normal (DNF - *Disjunctive Normal Form*), muito do processo de análise para produzir partições de teste pode ser automatizado. Estes últimos também descrevem uma técnica para extração automática de autômatos de estado finito a partir da especificação.

Teste de Partição de Categoria É uma técnica projetada para extrair um conjunto de testes funcionais de uma especificação. O método consiste em categorizar o domínio de entrada da unidade funcional que será testada com base nas principais características de entradas implícitas e explícitas da unidade. Geralmente, o método de partição de categorias é usado para gerar descrições do comportamento especificado quando se deseja realizar testes a partir de uma especificação informal. A descrição gerada é chamada janela (*frames*) de teste, que é similar a idéia de sub-domínio da técnica de partição da subseção anterior. Ostrand, em [OB88], mostra um método automatizado para gerar as janelas.

Como exemplo, deseja-se testar um método de uma classe implementada em C++ tendo como base sua representação formal em Z. Após aplicar a técnica sobre as declarações e esquemas da especificação, são obtidas as categorias (partições) e respectivas escolhas de valores de entrada para cada um dos métodos C++. A partir da descrição informal do método (janela), pode-se criar um ou mais valores para cada escolha dentro das categorias e criar o conjunto de testes para o método a ser testado.

Molde de Teste Em [SC93] é introduzido uma estrutura (*framework*) comum chamada Estrutura de Molde de Teste, denotada TTF (*Test Template Framework*), que utiliza a notação Z para realizar testes de caixa-preta. O molde de teste é um esquema Z que descreve um conjunto de casos de teste. Os casos de teste são gerados a partir dos moldes produzidos, que correspondem a uma partição do espaço de entrada. TTF fornece uma estrutura na qual 23 diferentes algoritmos de geração podem ser aplicados.

Teste de Mutação Em [CS94] está descrita a abordagem de teste denominada mutação de especificação. A idéia é inspirada pelo teste de mutação de programas (conforme apresentado na Seção 2.3), porém com diferentes propósitos. A especificação é alterada (mutantes), e para cada mutação, um teste é derivado para que seja distinguido o comportamento das especificações mutantes do comportamento da especificação original. O efeito é garantir que a IUT não implemente qualquer especificação incorreta. Os testes gerados pelos mutantes são testes negativos, isto é, quando a IUT implementa corretamente a especificação, o resultado esperado para execução dos testes é a falha.

4.3 Testes Formais Baseados em Máquinas de Estado Finitas

Uma máquina de estados finita, abreviada como FSM (*finite state machine*) [LY96], é uma 5-tupla (S, I, A, T, R), onde S é um conjunto de estados, I é o conjunto de estados iniciais ($I \subseteq S$), A é um conjunto representando o alfabeto de entrada, R é um conjunto representando o alfabeto de resposta, e T é um subconjunto do produto cartesiano $S \times A \times S \times R$ chamado relação de

transição. A interpretação dada para a relação de transição é que $(s,a,t,r) \in T$ se a máquina no estado s reagir a entrada a, movendo-se para o estado t e devolvendo como saída r. É dito que uma entrada a é valida em um estado s se existirem t e r tal que $(s,a,r,t) \in T$.

Nem sempre é possível observar algumas propriedades do estado interno da FSM, como por exemplo o valor de variáveis, pois tais elementos não fazem parte de uma FSM. Uma máquina de estados finita estendida, abreviada como EFSM (*extended finite state machine*), adiciona variáveis para cada estado. É comum na prática visualizar o conjuto de estados S como um subconjunto do produto cartesiano $D_1 \times D_2, ..., D_{n-1} \times D_n$, onde D_i é o domínio da i-ésima variável de estado x_i.

Um caso de teste abstrato gerado a partir de uma FSM consiste em uma seqüência de estímulos de entrada seguida pela resposta esperada, e o estado da FSM alcançado após o teste. No caso particular de EFSM, o teste possui os mesmos elementos do teste de FSM, com adição dos valores encontrados para as variáveis de estado alcançados após o teste.

As estratégias típicas para gerar testes a partir de FSM, objetivam detectar falhas de saída ou falhas de transferência [LY96]. Como exemplo de falhas de saída, podemos produzir testes cujas transições produzem saídas incorretas. Como exemplo de falhas de transferência, os testes possuem transições que levam para estados incorretos da IUT. Os testes gerados são baseados em uma relação de conformidade entre a especificação F e a implementação válida I [LPB93].

FSMs Completamente Especificadas e Determinísticas O método TT (*transition tour*) em [Pet01], produz um conjunto de testes que executam todas as transições da especificação e procura encontrar faltas de saída. O método D (*D-method*) verifica todas as transições para cada saída e procura detectar faltas de transferência.

FSMs Parcialmente Especificadas e Não-determinísticas FSMs parcialmente especificadas são adequadas para especificar sistemas que não cobrem todas as combinações de estados/entradas especificadas por uma FSM completamente especificada, por exemplo, protocolos de comunicação. Máquinas parcialmente especificadas podem levar a diversas interpretações devido a várias transições que estão ausentes, tais transições são chamadas: transições indefinidas, não principais (*non-core*), ou não importantes (*don´t care*). Essas transições podem ser consideradas implicitamente, o que significa que podemos completar as transições ausentes de uma forma tal a obter uma FSM que seja completamente especificada. Após completar a FSM, técnicas utilizadas para FSM completamente especificadas podem ser utilizadas de igual forma como foi apresentada na subseção anterior. Outra visão para as transições faltosas é considerar que elas nunca serão executadas pela implementação, portanto deve-se evitá-las. Essa abordagem leva para geração de um conjunto de testes a partir de uma FSM parcialmente especificada no contexto das FSMs não-determinísticas.

Várias estratégias de teste têm sido desenvolvidas para geração de testes a partir de FSM não-determinística (parcial), entre elas o método Wp (*Wp-method*) [LvBP94].

FSMs Comunicantes Muitos sistemas são naturalmente modelados por um conjunto de FSMs ao invés de uma única máquina. Essas máquinas operam concorrentemente e trocam

mensagens em uma maneira similar a SDL [BH89] (é linguagem gráfica para especificação e projeto cujo modelo é baseado em EFSM). FSMs que podem se comunicar e cooperar dessa forma são chamadas FSMs comunicantes (CFSMs). Tal comunicação é suportada por filas e canais.

Um sistema especificado como CFSM pode ser transformado em uma FSM equivalente, e gerados os testes utilizando os métodos existentes para FSM não comunicantes. Essa transformação ocorre através de uma análise exaustiva das possibilidades de interação (*exhaustive reachability analysis*), que é possível quando as filas e canais possuem tamanhos finitos. Para evitar a explosão combinatória, alguns métodos procuram testar as transições de cada componente da FSMs de forma individual [Hie01].

4.4 Testes Formais Baseados em LTS

Álgebras de processo como LOTOS[psI89], CCS[Mil89], e CSP[Hoa85] provêem um formalismo elegante para especificar comunicação entre entidades denominadas processos (mais detalhes no Capítulo 3). LTS (como apresentado na Seção 3.2) descreve a semântica operacional das álgebras de processo. Como não existe abordagem conhecida para geração de casos de teste baseada puramente em representações algébricas, os trabalhos relacionados a geração de testes a partir de álgebras de processo trabalham basicamente a nível de LTS. Esta seção aprofunda e mostrada detalhes sobre as técnicas de teste formais baseadas em LTS. Os conceitos apresentados nesta seção são utilizados como fundamentação teórica para a abordagem de geração proposta no Capítulo 5.

Na literatura de teste, testar a partir de especificações LTS consiste na interação entre o agente formal representando a IUT e outro representando o caso de teste (TC — *test case*), ambos descritos em termos de LTSs [WQ96]. A existência de tal especificação formal da IUT é denominada hipótese de teste [Gau95] [Ber91]. Para que os testes sejam gerados de maneira coerente, é preciso estabelecer a noção de conformidade com a especificação. Conformidade é definida por meio de uma relação de implementação (conceito semelhante ao da conformidade entre máquinas de estado, ver Seção 4.3) entre o modelo formal da IUT e a especificação formal do comportamento do sistema a ser testado (SUT — *system under test*) [Tre96c] [WQ96]. No contexto específico de LTS, onde $LTS(L)$ denota o conjunto de LTSs que podem ser construídos considerando o alfabeto de eventos L, a relação de implementação é da forma: $\mathbf{imp} \subseteq LTS(L) \times LTS(L)$. Uma implementação $i \in LTS(L)$ está em conformidade com uma especificação $s \in LTS(L)$ se exatamente se i **imp** s.

O TC modela o comportamento do testador durante um a execução do teste realizado sobre a IUT. Todo teste deve durar um tempo finito, portanto um caso de teste deve ter um comportamento finito. No mais, o testador deve possuir o máximo de controle sobre o processo de teste, portanto, o caso de teste deve ser determinístico.

Para que seja possível decidir sobre o sucesso da execução de um teste sobre uma IUT, cada estado do caso de teste possui um veredito associado. Existem basicamente dois veredictos para a execução de um teste: passou (a IUT passou na execução do teste), e falhou (um comportamento errado foi observado e a IUT é considerada falha). Existe ainda a noção de

teste inconclusivo, geralmente utilizado quando o teste é dirigido a objetivo [JJ05]. O veredito inconclusivo acontece quando, durante os testes, o comportamento da IUT observado é consistente nos termos da relação de implementação, no entanto, o objetivo do teste não pode ser alcançado.

Formalmente um caso de teste é um LTS que possui um mapeamento associando que associa vereditos a estados, onde v (denota um veredito) e S (denota um estado).

$$v : S \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$$

Esse mapeamento é denominado função de veredito. A execução de um caso de teste se comporta de acordo com a sua execução paralela síncrona com a implementação [2]. Nesse contexto, o paralelismo considera as representações formais do teste e da implementação. Colocando o caso de teste t em paralelo com a implementação a ser testada p, temos : $t \parallel p$. Essa combinação de comportamentos via paralelismo ocorre até que não seja mais possível executar interações comuns entre TC e IUT, nesse caso, ocorre um *deadlock*. O *deadlock* ocorre quando o caso de teste alcança o estado final, ou quando as ações propostas pelo caso de teste não podem ser aceitas pela implementação. Em qualquer dessas situações, quando o teste entra em *deadlock* no seu estado s com algum estado da implementação, o resultado da execução corresponde ao veredito $v(s)$.

Uma IUT falha em um caso de teste se pelo menos uma execução de teste levar ao veredito *falhou*. Ele passa o teste, se somente se, todas as possíveis execuções levam para o veredito *passou*. Notemos que, como o comportamento da implementação IUT pode ser não-determinístico então, diferentes rodadas de um mesmo teste podem levar a diferentes veredictos. O veredito final só é alcançado após múltiplas execuções do mesmo teste sobre a implementação. Caso nenhuma heurística ou critério de parada seja empregado para delimitar a quantidade máxima de execuções do experimento de teste, o teste deverá executar infinitas vezes (uma possibilidade que é obviamente inviável).

Como exemplos de especificações utilizadas em boa parte desta seção, temos na Figura 4.4 os LTSs que modelam o comportamento dos processos p, $p1$ e $p2$ (estes processos só estão definidos operacionalmente). Cada um desses processo especifica o funcionamento de uma máquina de café. O conjunto L de ações possíveis nessas especificações é $\{shil, choc, liq\}$. A ação *shil* representa o ato "apertar botão", e *liq* e *choc* capturam respectivamente os eventos "leite servido"e "chocolate servido". O funcionamento das duas primeiras máquinas (p e $p1$) é semelhante, aperta-se o botão (*shill*) e a máquina serve umas das duas opções de bebida (leite *liq* ou chocolate *choc*) de acordo com a opção selecionada pelo usuário. Já a terceira máquina ($p2$), depois que o usuário aperta o botão, a máquina servirá uma das duas bebidas de forma aleatória (não-determinística), sem que o usuário tenha controle sobre qual será a bebida servida. Isto acontece porque $p2$ oferece dois eventos com o mesmo nome que levam a caminhos diferentes (mesma situação descrita na Seção 3.1.4 que mostra como uma escolha externa pode se comportar com uma escolha interna).

[2] Alguns conceitos relativos à algebras de processo (mais particularmente CSP) como: processo, paralelismo, traces, *deadlock*, *livelock* não-determinismo e LTS, utilizados nessa Seção, estão melhor explicitados no Capítulo 3.

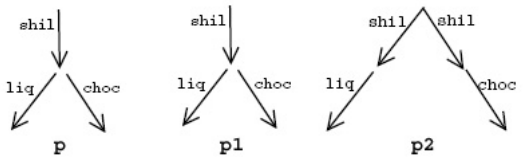

Figura 4.4 LTSs p, p1 e p2

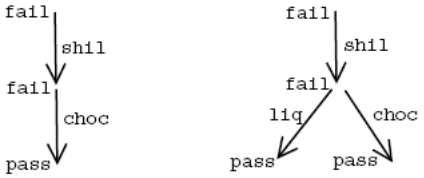

Figura 4.5 Casos de teste t1 e t2

Já como exemplos de casos de teste LTS, temos na Figura 4.5 os testes $t1$ e $t2$. O teste $t1$ especifica que se a IUT não comunicar nenhum evento e entrar em *deadlock*, então o estado inicial do teste leva ao veredito falha (*fail*). Já se o usuário apertar o botão (*shil*) e a IUT entrar em *deadlock,* então o estado do teste também leva ao veredito falha. Porém, se o usuário da IUT apertar o botão (*shil*) e em seguida escolher chocolate (*shoc*) e a IUT entrar em *deadlock*, o estado do teste leva ao veredito passou (*pass*). O teste $t2$ especifica que se a IUT não comunicar nenhum evento e entrar em *deadlock*, então o estado inicial do teste leva ao veredito falha (*fail*). Caso o usuário apertar o botão (*shil*) e a IUT entrar em *deadlock*, então o estado do teste também leva ao veredito falha. Porém, se o usuário da IUT apertar o botão (*shil*) e em seguida escolher chocolate (*shoc*), ou, escolher leite (*liq*) e a IUT entrar em *deadlock*, o estado do teste leva ao veredito passou (*pass*).

Usando como exemplo os processos da Figura 4.4 e os testes da Figura 4.5, podemos perceber que o processo $p2$ pode falhar em uma execução com o teste *t1*, e passar em uma execução com o teste *t2*. $p2$ pode falhar com *t1* porque, em um cenário particular de execução de $p2$ onde ocorre o trace de *deadlock* $\langle shil, liq \rangle$, onde teste é levado para um estado cujo veredito é falha. No entanto, executando o teste *t2* com o mesmo processo, ocorre o trace de *deadlock* $\langle shil, choc \rangle$, que leva o teste a um estado cujo veredito é passou. $p2$ falha em *t1* porque existe uma possível execução que irá falhar. $p2$ passa com *t2* pois, todas as possíveis execuções de $p2$ com esse teste ($\langle shil, liq \rangle$ e $\langle shil, choc \rangle$) levam ao veredito pass.

Algoritmos de geração precisam ser desenvolvidos para produzir um conjunto de testes a partir de uma dada especificação. Formalmente, um algoritmo de geração de testes pode ser descrito por uma função:

$$gen_{\mathbf{imp}} : LTS(L) \rightarrow \mathbb{P}\, LTS(L) \tag{4.10}$$

Um conjunto de testes gerados $gen_{\mathbf{imp}}(s)$ testa a conformidade das implementações com respeito à especificação s e à relação de implementação **imp**. Em um cenário ideal, o conjunto

de testes pode distinguir entre implementações que estão em conformidade das que não estão. A implementação deve passar no conjunto de testes se apenas se a implementação estiver em conformidade com **imp**. Nesse caso, o conjunto de teste é chamado completo (*complete*) [WQ96]. Infelizmente, essa característica não pode ser obtida na prática pois na maioria dos casos um conjunto de testes completo possui infinitos testes. Portanto, o mais comum é considerar requisitos mais fracos como: coerente (*sound*) e exaustivo (*exhaustive*). As Equações (4.11) e (4.12) mostram, respectivamente, as implicações de um caso de teste que é coerente e de um caso de teste que é exaustivo.

$$\forall IUT \bullet IUT \ \ \mathbf{imp} \ \ s \Rightarrow \neg (TC \ \ pode rejeitar \ \ IUT) \tag{4.11}$$

$$\forall IUT \bullet \neg (IUT \ \ \mathbf{imp} \ \ s) \Rightarrow (TC \ \ pode rejeitar \ \ IUT) \tag{4.12}$$

Durante a execução de um conjunto de testes que é coerente, todas as implementações que são conformes e, algumas não conformes, passam: toda implementação que falha é não conforme, entretanto, nem toda implementação que passa é conforme. Já a execução de um conjunto que é exaustivo, todas as implementações não conformes e, algumas conformes, falham: qualquer implementação que não é conforme falha, porém nem todas que falham são não conformes.

4.4.1 Relação de Implementação

Casos de teste são utilizados para determinar se uma implementação está conforme com a especificação. No entanto, dependendo do tipo do teste, a quantidade de observações de uma IUT realizadas por um TC pode variar. Essa quantidade de observações entre TC e a IUT é chamada relação de implementação.

Na literatura de testes formais [WQ96, Tre96c, Tre99] o teste é dado como a interação de dois agentes: a IUT p e o teste t. Imaginamos portanto, a especificação e o sistema como LTSs. Considerando as relações de implementação mais simples, onde o sistema e o TC se comunicam de maneira síncrona, a especificação p pode realizar um evento e se apenas se o teste t realiza o mesmo evento e. Essa comunicação síncrona é modelada usando o operador de paralelismo síncrono || entre processos, cuja semântica em LTS corresponde a equação (3.1). Os cenários de teste são então analisados a partir das observações do comportamento de $p \parallel t$. Uma dessas observações se chama trace; que é um conjunto (tanto finito ou infinito) de seqüências realizadas por um LTS. Na Figura 4.4, temos que:

$$traces(p) = traces(p1) = traces(p2) = \{\langle\rangle, \langle shil\rangle, \langle shil, choc\rangle, \langle shil, liq\rangle\}$$

Para considerar traces com e sem *deadlock* usamos duas outras funções: $obs'(t,p)$ e $obs(t,p)$. A função $obs'(t,p)$ denota todos possíveis traces observados em $p \parallel t$, enquanto $obs(t,p)$ denota os traces onde ocorrem *deadlocks*. Considerando o processo p como um experimento de teste, e $p2$ como a implementação a ser testada, $p \parallel p2$ pode entrar em deadlock tanto em $\langle shil, liq\rangle$ quanto em $\langle shil, choc\rangle$. Portanto:

$$obs(p,p2) = \{\langle\rangle, \langle shil, liq\rangle, \langle shil, choc\rangle\}$$

$obs'(p, p2) = \{\langle\rangle, \langle shil\rangle, \langle shil, liq\rangle, \langle shil, choc\rangle\}$

Nas Subseções seguinte descrevemos brevemente algumas das relações de implementação mais conhecidas.

4.4.1.1 A relação *TracePreorder*

Uma IUT representada por um LTS p está corretamente implementada com relação à especificação s (LTS), se os traces de p são iguais ou menores que os traces de s. Formalmente, tem-se a definição:

Definição 4.4.1. *Seja* $p, s \in LTS(L)$ *então* $p \leq_{tr} s \stackrel{def}{=} traces(p) \subseteq traces(s)$

A Definição (4.4.1) introduz uma relação de conformidade denominada *trace preorder* [JW90]. Dados dois LTSs p e s com mesmo conjunto de eventos L, p é um trace preorder de s ($p \leq_{tr} s$) se somente se $traces(p) \subseteq traces(s)$.

Usando como exemplo os LTSs p, $p1$ e $p2$ temos que $p1 \leq_{tr} p$ e $p2 \leq_{tr} p$. Apesar de $p2 \leq_{tr} p$ (a IUT $p2$ implementa a especificação p com respeito ao trace preorder), escapa das observações de $\leq_{tr}$ um incoveniente com os qual os usuários da implementação podem se deparar: p especifica que depois de apertar o botão (*shil*) o usuário tem a opção de escolher entre servir leite (*liq*) ou chocolate (*choc*), entretanto, a implementação $p2$ pode recusar-se a fornecer uma dessas opções. Na implementação, após apertar o botão (*shil*), a máquina de café faz uma escolha não-determinística entre oferecer leite (*liq*) ou chocolate (*choc*). Supondo que a máquina escolha oferecer *liq* e o usuário faz a escolha por *choc*, essa situação de impasse caracteriza um deadlock: nenhuma bebida será servida pois, não haverá interações posteriores. Isso acontece porque a relação $\leq_{tr}$ considera apenas as seqüências de ações observadas; não leva em conta como são resolvidas as escolhas especificadas no comportamento de s: tanto o ambiente externo quanto a máquina internamente podem fazer a escolha.

4.4.1.2 A relação *TestingPreorder*

Testing preorder [Nic87] é uma relação de implementação mais forte e sofisticada que trace preorder. Em complemento a trace preorder, além de observar que os possíveis traces da implementação devem estar contidos nos traces da especificação, requer que todo deadlock observado por um usuário (ou testador) na implementação também seja um deadlock encontrado na especificação. É uma relação que captura qualquer discrepância que possa ser observada no LTS por um observador externo em outro LTS. Formalmente é definida:

Definição 4.4.1. *Seja* $p, s \in LTS(L)$ *então*
$p \leq_{te} s \stackrel{def}{=} \forall u \in LTS(L) : obs(u, p) \subseteq obs(u, s) \wedge obs'(u, p) \subseteq obs'(u, s)$

A Definição (4.4.1) introduz a relação de conformidade testing preorder. Sejam p (representa uma implementação) e s (representa a especificação) dois LTSs que possuem o mesmo alfabeto L, e seja o LTS u (de mesmo alfabeto) a representação de um observador: p é testing preorder de s exatamente se, todas as observações de traces (*obs* e *obs'*) realizadas por u na

implementação são subconjuntos das que podem ser realizadas por u na especificação, isto é, $obs(u,p) \subseteq obs(u,s)$ e $obs'(u,p) \subseteq obs'(u,s)$.

Usando como exemplo os LTSs p, $p1$ e $p2$ temos que $p1 \leq_{te} p2$, porém $p2 \not\leq_{te} p1$. O LTS $p1$ é uma implementação válida para $p2$ pois, todos os traces e deadlocks de $p1$ podem ser observados em $p2$. No entanto, o contrário não é verídico: $p2$ não é implementação válida para $p1$, pois o conjunto de deadlocks observado em $p2$ não está contido nos de $p1$. Os traces e os deadlocks observados em $p1$ correspondem a $obs'(u,p1) = \{\langle\rangle, \langle but\rangle, \langle but, liq\rangle, \langle but, choc\rangle\}$, e $obs(u,p1) = \{\langle but, liq\rangle, \langle but, choc\rangle\}$. Traçando um paralelo entre $p1$ e $p2$ pode ser observado que $obs'(u,p2) \subseteq obs'(u,p1)$, no entanto $obs(u,p2) \not\subseteq obs(u,p1)$. Após apertar o botão *but*, o usuário pode escolher *liq* ou *choc*, enquanto o não-determinismo da máquina escolhe uma opção diversa do usuário, o que resulta em um deadlock em $\langle but\rangle$. Como resultado, $obs'(u,p2) = \{\langle but\rangle, \langle but, liq\rangle, \langle but, choc\rangle\}$.

4.4.1.3 A relação **conf**

A relação **conf** [Bri88] é uma relação de implementação similar a $\leq_{tr}$. É uma modificação de $\leq_{tr}$ pela restrição das observações consideradas para apenas os traces que estão contidos na especificação, o que torna a tarefa de teste mais flexível: apenas os traces da especificação precisam ser considerados, e não o gigantesco conjunto de todas observações, que muitas vezes são traces não especificados. De outra forma, **conf** requer que uma implementação faça o que ela deve fazer, e não o que ela não faz e o que ela não está habilitada a fazer. Esse relaxamento da relação permite que a implementação contenha, de forma adicional, comportamentos que não foram especificados: de maneira que, não seja ferida a conformidade com a especificação.

Definição 4.4.1. *Seja $i, s \in LTS(L)$ então*

$$i \ \textbf{\textit{conf}} \ s \stackrel{def}{=} \forall u \in LTS(L): \quad (obs(u,i) \cap traces(s)) \subseteq obs(u,s) \ \wedge \ (obs'(u,i) \cap traces(s)) \subseteq obs'(u,s)$$

A Definição (4.4.1) caracteriza que, dados dois processos i e s com mesmo conjunto de eventos L, temos a relação de conformidade i **conf** s, se somente se para cada caso de teste t com o mesmo conjunto de eventos L temos que: (1) $(obs(t,i) \cap traces(s)) \subseteq obs(t,s)$ e (2) $(obs'(t,i) \cap traces(s)) \subseteq obs'(t,s)$. Destacando que, as duas restrições consideram apenas os traces encontrados na especificação. A restrição (1) declara que sempre que um caso de teste t entra em deadlock quando interage com a implementação i, depois do trace $\sigma \in traces(s)$, então t pode entrar em deadlock depois do trace σ quando interagindo com a especificação s.

Como exemplo, supondo $p2$ a implementação de $p1$ e, observando o comportamento da execução do teste $t1$ em $p2$ ($t1 \parallel p2$). No contexto dessa execução, o trace $\langle shil\rangle \in traces(p1)$ pode entrar em deadlock. O deadlock em $\langle shil\rangle$ ocorre devido ao não-determinismo de $p2$: que tanto pode ir para *liq* ou *choc* antes de sincronizar com *choc* que faz parte da continuação do teste. De acordo com **conf** , uma das condições para que $p2$ seja implementação de $p1$ no trace $< shil >$ é, que $(obs(t1,p2) \cap \langle shil\rangle) \subseteq obs(t1,p1)$. Observando esse trace com $p1$, vemos que o mesmo não entra em deadlock, portanto $obs(t1,p2) \not\subseteq obs(t1,p1)$, o que invalida a regra (1). Como conseqüência, $\neg(p2 \,\textbf{conf}\, p1)$. Já considerando $p1$ como uma implementação de $p2$, temos que $(p1 \,\textbf{conf}\, p2)$. Isso acontece pois, as observações dos traces $p1$ quando $p2$ é implementação estão de acordo com as regras (1) e (2).

4.4.1.4 A relação **ioco**

Outra relação de implementação bastante difundida é a relação **ioco** (Input-Output Conformance) [Tre96b], que representa uma evolução com relação a **ioconf** [Tre96c]. A diferença primordial entre **conf** e **ioco** é que a primeira é baseada em comunicações síncronas entre os LTSs que representam TC e a IUT, enquanto a segunda se baseia em IOLTSs. IOLTS (input-output transition systems) é uma classe especial de LTS onde o conjunto de eventos L é particionado em eventos de entrada L_I e de saída L_U. Além disso, requer que todas as ações de entrada estejam sempre habilitadas em cada estado s: $?a \in L_I : s \xrightarrow{?a} s'$. Nesse formalismo, as entradas de um IOLTS se comunicam com as saídas de outro(s) IOLTS(s), e vice-versa. Em um contexto particular, onde o testador é representado por um caso de teste IOLTS, as ações de saída do testador são as ações de entrada para o IOLTS da IUT, e as saídas da IUT são entradas para o teste. **ioco** considera como hipótese de teste que a implementação possa ser especificada via um IOLTS.

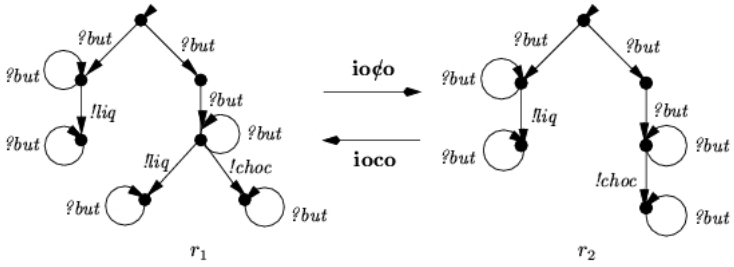

Figura 4.6 Sistemas de Transições Rotuladas (LTS)

Na Figura 4.6 podemos ver os LTSs $r1$ e $r2$ que correspondem a especificações LTS para duas diferentes máquinas de café. Essas especificações são IOLTS onde cujo alfabeto de entrada corresponde ao conjunto $L_I = \{?but\}$ e $L_U = \{!choc, !liq\}$ ao alfabeto de saída. O prefixo ? antes do nome do evento denota uma ação de entrada, e ! denota uma ação de saída. Esses LTSs possuem como estado inicial o nó superior do grafo (que está apontado por seta). Note que em cada estado sempre está habilitado a ocorrência de um evento de entrada, representado pelo loop ?*but* do estado para ele próprio. Em $r1$, pode-se obter leite apertando uma vez o botão, trace $\langle ?but, !liq\rangle$ e, chocolate ou leite após apertar o botão duas vezes consecutivas, traces $\langle ?but, ?but, !choc\rangle$ e $\langle ?but, ?but, !liq\rangle$. O LTS $r2$ tem comportamento idêntico a $r1$ quando o botão é apertado apenas uma vez, já quando o botão é apertado duas vezes, produz apenas leite.

A relação **ioco** define uma relação de implementação entre uma especificação s e sua implementação i, ambos expressos como IOLTS:

Definição 4.4.1. *Sejam $i, s \in IOLTS(L)$ então,*

$$i \ \textbf{\textit{ioco}} \ s =_{def} \ \forall \sigma \in Straces(s),\ Out(\Delta_i \ after \ \sigma) \ \subseteq \ Out(\Delta_s \ after \ \sigma)$$

A relação **ioco** considerada um conjunto de observações $\sigma \in Straces$ denominado *suspension trace* que representa todos os comportamentos visíveis pelo ambiente: seqüências de entradas e saídas incluindo eventos de *quiescence*. Quiescence (denotado pela constante δ) é um

evento de saída que representa a ausência de comportamento. Deadlock, livelock e ausência de saída (outputlock) são exemplos de situações cujo comportamento é modelado como δ.

A Definição (4.4.1) caracteriza que uma implementação *i* é **ioco** com relação a uma especificação *s* se, qualquer evento de saída produzido pela implementação depois de um *suspension trace*, também pode ser produzido pela especificação depois do mesmo trace σ. Os símbolos Δ_i e Δ_s representam o comportamento visível da implementação e da especificação. O símbolo Δ_{lts}, onde *lts* é um LTS qualquer, representa o comportamento de *lts* modificado pela adição da transição δ para cada estados do LTS onde se caracteriza quiescence. Por outro ângulo, para *i* **ioco** *s* ser verdade, não existe qualquer evento de saída após o suspension trace σ da implementação *i* que não seja encontrado em *s* após se comportar com o mesmo trace σ.

Na Figura 4.6 temos que r_2 ***ioco*** r_1, mas $\neg(r_1$ ***ioco*** $r_2)$; o LTS r_1 não é uma implementação **ioco** válida para r_2 desde que, r_1 produz a saída $!choc$ depois do trace $\langle ?but, \delta, ?but\rangle$. Essa saída pertence ao conjunto de saídas de r_1, $!choc \in \text{Out}(r_1 \text{ after } \langle ?but, \delta, ?but\rangle)$, porém, não pode ser encontrada no conjunto de saídas de r_2 para o mesmo trace, $!choc \notin \text{Out}(r_2 \text{ after } \langle ?but, \delta, ?but\rangle)$.

4.4.2 Geração de Testes

A literatura possui um número de relações de implementação e uma gama de algoritmos de geração de testes.

Tretmans define em [Tre96c] um algoritmo de geração de testes que a partir de especificações LTS gera um conjunto de testes coerente, de acordo com a relação de implementação **conf** . Dado o processo $p1$ da Figura 4.4, temos o conjunto de testes CT especificado na notação de CSP: $CT = T1;\ T2$. Vendo de maneira isolada os testes $T1$ e $T2$, que possuem comportamentos complementares e são executados seqüencialmente; temos o teste $T1$ cuja especificação é $shil \rightarrow Skip$, que verifica se a implementação consegue executar o evento *shil* a partir do estado inicial. Outro teste $T2$ corresponde a especificação $(choc \rightarrow Stop \,\Box\, liq \rightarrow Stop)$, que verifica se após executar *shil* a implementação pode executar a opção *choc* ou *liq*, de forma que o ambiente externo (o usuário) tenha controle sobre a escolha das opções.

4.5 Ferramentas de Geração Baseadas em Modelos de Comportamento

Nessa seção é feita uma revisão das principais ferramentas utilizadas na geração de testes a partir de modelos de comportamento do sistema.

Apesar de uma parte significante da indústria de telecomunicações, aeroespacial e microeletrônica terem experimentado a utilização de modelos para verificação e geração de testes por mais de uma década. A indústria de tecnologia da informação ainda está engatinhando quando se trata da utilização de ferramentas de geração de casos de teste baseadas em modelos.

As ferramentas que serão listadas como geradoras de teste a partir de modelos aceitam duas principais entradas:

- Um modelo formal do software que será testado, e;
- Um conjunto de diretivas de geração de testes que guiam a ferramenta em sua tarefa de geração.

Como visto anteriormente na Seção 4.1.2, existem critérios de seleção explícitos e implícitos. As diretivas de geração são critérios explícitos de seleção fornecidas pelo usuário da ferramenta de geração, como exemplo de diretiva de seleção, citamos os propósitos de teste [WQ96]. Em muitas ferramentas, as diretivas de geração de testes são critérios implícitos que fazem parte da estrutura de geração da ferramenta. A saída de tais ferramentas é um conjunto de casos de teste, que incluem uma seqüência de estímulos fornecidos como entrada para a IUT, e as respostas esperadas para os mesmo estímulos, como predito pelo modelo.

4.5.1 Ferramentas do Projeto AGEDIS

De maneira especial, podemos citar a iniciativa AGEDIS (Automated Generation and Execution of Test Suites for Distributed Component-based Software) [HN04]. AGEDIS foi um projeto de pesquisa de três anos (de 2000 a 2003), com o objetivo de criar ferramentas que dessem suporte a geração e execução automática de conjunto de testes para software baseados em componentes distribuídos. O projeto foi desenvolvido a partir do consórcio de várias instituições como: Laboratório de pesquisa da IBM em Haifa (que gerenciou o projeto), Laboratório de Computação da Universidade de Oxford, Laboratório de Verimag da Universidade Joseph Fourier, France Telecom R&D, laboratório de desenvolvimento da IBM na Inglaterra, Intrasoft International, entre outros. Para mais informações acessar a página WEB do projeto `http://www.agedis.de/`.

Como uma contribuição central do AGEDIS, destaca-se a arquitetura de engenhos e interfaces para geração e execução de testes da Figura 4.7, que implementa os requisitos:

- Gerar casos de teste automaticamente;
- Analisar/editar os casos de teste gerados;
- Simular a execução desses casos de teste;
- Executar os casos de teste na aplicação real;
- Armazenar e analisar o trace da execução dos casos de teste; e,
- Re-gerar os casos de teste novamente de acordo com os critérios de adequação alcançados e das falhas que mais ocorreram durante a execução.

A arquitetura foi projetada para ser modular e ter interfaces bem definidas, permitindo que outras empresas e universidades possam utilizar essa arquitetura da forma mais flexível possível. É possível acoplar quaisquer ferramentas na arquitetura, desde que sejam obedecidas as interfaces.

Voltando a Figura 4.7, pode-se observar elementos anotados como elipses e quadrados. As elipses contornadas representam as entradas fornecidas pelo usuário (user input) e acessadas via interface gráfica. As elipses sem contorno são as interfaces públicas (public interfaces). As interfaces públicas correspondem aos dados intermediários utilizados como comunicação entre os componentes da arquitetura. Dados intermediários podem ser editados e visualizados por

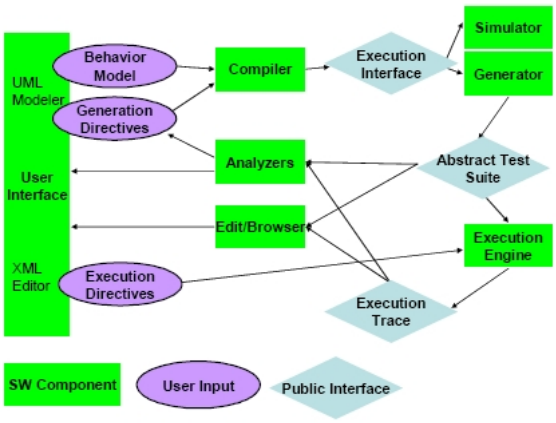

Figura 4.7 Arquitetura AGEDIS

meio das ferramentas da própria arquitetura. Os quadrados são os componentes de software que processam as entradas e interfaces públicas.

O modelo de comportamento (behaviour model) do programa a ser testado, é escrito em AML (AGEDIS Model Language) que possui semelhanças com UML. Editores de UML são usados para criar os diagramas de classes, estados e objetos que vão representar o comportamento do sistema. Detalhes do funcionamento dinâmico do modelo são modelados com a linguagem IF (intermediate format) [BGM]. IF A ferramenta CASE adotada é Objectering UML - a única cujo modelo pode ser lido pelo compilador.

Depois do modelo de comportamento, a segunda entrada para geração dos teste são as diretivas de geração (generation directives). Diretivas são definidas utilizando gráficos e wild cards (templates do formato do caso de testes). Correspondem às estratégias e táticas de teste dos usuários. A diretiva de geração pode ser utilizada para definir o critério de cobertura utilizado. A ferramenta dispõe vários tipos de cobertura já implementados como diretivas: cobertura dos estados e cobertura das transições. Existe também a opção de escolher a cobertura de elementos do modelo que estão anotados com estereotipos específicos. Outra possibilidade para diretiva de teste é: o tamanho dos casos de teste, que limita o número de procedimento que compõe cada teste gerado.

Definido o modelo de comportamento e as diretivas de geração, é possível obter casos de teste a partir do gerador (generator), como também simular a execução dos testes utilizando o simulador (simulator). Com o simulador é possível fornecer entradas arbitrárias para o modelo do comportamento e verificar as saídas produzidas. Nesse ponto da arquitetura, o artefato produzido de maior relevância é o grupo de testes abstratos (abstract test suit). Testes abstratos são arquivos XML que possuem seqüências de entradas fornecidas à IUT, e as saídas esperadas; podem ser visualizados com ferramentas específicas, ou editados utilizando editor de texto comum.

Casos de teste abstratos não estão prontos para executar na IUT, pois os dados e eventos dos testes precisam ter uma ligação com a tecnologia específica que será utilizada para rodas os testes na IUT. É preciso de um elemento concreto que realmente possa interagir com o programa

implementado em uma linguagem de programação específica, como exemplo, um tipo inteiro existente no caso de teste abstrato pode ser mapeado como o tipo int de Java no momento em que precisar ser executado. Esses componentes de concretização são chamados de diretivas de execução de testes (test execution directives): arquivos XML que fazem a relação dos elementos do modelo (abstratos) com elementos reais da linguagem de programação da aplicação (concretos). O engenho de execução (execution engine) utiliza as diretivas de execução junto com o conjunto de testes abstratos para executar testes sobre a IUT. As informações coletadas durante a execução são persistidas no registro de execução (execution trace). O registro de execução possui o mesmo formato de um caso de teste (mesmo esquema XML), de maneira a facilitar a comparação entre os testes e as respostas encontradas durante a execução.

O analisador (analyzer) pode ser usado tanto para analisar a cobertura da execução, tanto para gerar novos casos de teste. O analisador lê o registro de execução e fornece informações de quais métodos e valores do modelo não foram cobertos pelos testes durante a execução. A partir dessas informações, pode ser gerada uma nova versão das diretivas de geração cujos testes resultantes possuiram uma maior cobertura do modelo. O analisador pode ainda agrupar casos de teste que falham para que se tornem apenas um, isso pode ser útil para re-testar o sistema após uma correção de bug. Essa técnica torna mais eficiente a forma de verificar a correção de um bug específico, pois é utilizada uma quantidade menor de testes.

4.5.2 Ferramentas Comerciais

Nas subseções seguintes estão descritas brevemente ferramentas comerciais baseadas em modelos de comportamento.

4.5.2.1 TVGS

TVGS é a abreviação para Test Vector Generation System (Sistema Gerador de Teste de Vetor) [TVE06]. Essa ferramenta aceita modelos de requisitos e de comportamento do sistema descritos em uma linguagem proprietária chamada T-VEC Linear Form (TLF). É utilizada principalmente na indústria aeroespacial, e não possui qualquer suporte a UML ou qualquer outra linguagem de modelagem largamente difundida. O produto vem com um ambiente gráfico para criação de modelos SCR [HKL97] (uma notação para modelagem de requisitos de software baseada em tabelas proprietária).

A diretiva de geração se encontra implícita no algoritmo utilizado para gerar os testes, e como resultado da geração são produzidos casos de teste abstratos que incluem as saídas esperadas. Os testes gerados cobrem os valores limites no caminho de decisão (um tipo de cobertura de condicional, ver 4.1.2).

Essa ferramenta possui uma interface para traduzir os casos de teste abstratos para scripts de teste que podem ser executados automaticamente.

4.5.2.2 Conformiq Test Generator

Conformiq Test Generator [Con06] é uma ferramenta produzida pela Conformiq Software Ltda. Essa ferramenta aceita como entrada diagramas de estado (statecharts) de UML como o modelo

do SUT, que incluem especificações com propriedades de tempo real.

O principal produto da análise do gerador é um simulador de comportamento que pode ser utilizado de várias formas diferentes:

- Modo de lote (batch mode) - Utilizado para gerar casos de teste usando a notação TTCN, os casos de teste gerados podem ser executados contra a IUT em um momento posterior. Os casos de teste possuem resultados esperados e veredictos.
- Modo Interativo - Permite executar o gerador de testes de forma sincronizada com a IUT. O comportamento do sistema é verificado em tempo de geração.
- Modo de Trace - Uma atividade do sistema pode ser modelada por um trace, que pode ser rodado através do gerador de testes para verificar se suas entradas e saídas estão conforme ou não, com respeito à especificação.

TTCN (Testing and Test Control Notation) [SW92] é um padrão da ETSI (European Telecommunications Standards Institute) para casos de teste automáticos que está na sua terceira versão, TTCN-3. Esse padrão dá suporte a testes caixa-preta automatizados para sistemas reativos distribuídos em tempo real, permitindo múltiplas visões para os testes: 1) notação textual núcleo, no estilo de linguagem de programação; 2) representação gráfica, e, 3) representação tabular. O padrão inclui a descrição padronizada das interfaces de teste para executar casos de teste TTCN-3 em ambientes distribuídos, suportada pelos múltiplos fabricantes independentes de ferramentas para telecomunicações.

Essa ferramenta utiliza algoritmos de exploração junto com técnicas de medida de cobertura para selecionar os testes a serem gerados, a página WEB da ferramenta não fornece detalhes adicionais sobre os critérios utilizados.

4.5.2.3 Tau TTCN Suite

Tau TTCN Suite [Tel06a], corresponde a um conjunto de várias ferramentas da Telelogic utilizadas para criação, simulação e manipulação de modelos SDL [BH89] e casos de teste TTCN-2 (TTCN versão 2). É capaz de gerar scripts de teste TTCN a partir dos modelos SDL. Este conjunto de ferramentas é um ambiente padrão para teste de conformidade de sistemas de comunicação, é largamente utilizado para testar equipamentos de telecomunicação desde chips de comunicação embutidos até gigantescos aparelhos de *switch* e serviços de rede inteligentes. Vem com um compilador TTCN-2 que compila os scripts de teste TTCN-2 em código C executável. A literatura disponível para essa ferramenta não indica o critério de seleção empregado na geração dos testes.

4.5.2.4 Autolink e TestComposer

Autolink e TestComposer [SEG00] são duas ferramentas de geração bastante integradas com seus ambientes de desenvolvimento. TestComposer é construída em cima da ferramenta de modelagem ObjectGeode [Ver06]. Autolink é parte da ferramenta Telelogic Tau [Tel06b]. Ambas as ferramentas são utilizadas para procurar erros dinâmicos e inconsistências em especificações

SDL, e gerar conjunto de testes TTCN a partir das especificações SDL e propósitos de teste. Propósitos de teste podem ser definidos:

- Manualmente, através de um editor de MSC (Message Sequence Charts) [Soc06];
- Interativamente através de um simulador passo a passo do sistema especificado em SDL; e,
- Totalmente automático.

4.5.3 Ferramentas Proprietárias

Nas subseções seguintes estão descritas brevemente ferramentas proprietárias baseadas em modelos de comportamento.

4.5.3.1 GOTCHA-TCBeans

GOTCHA-TCBeans [FHNS02] é uma ferramenta ancestral da ferramenta de geração e execução de testes do projeto AGEDIS [HN04]. A linguagem de modelagem utilizada é a GOTCHA, uma extensão de MurΦ [Dil96]. A linguagem MurΦ é extendida pela adição de diretivas de geração de testes que permitem a especificação de projeções arbitrárias do espaço de estados que são utilizadas como critério de cobertura. Os casos de teste incluem resultados esperados. A estrutura de tradução é provida pelo TCBeans. Essas ferramentas têm sido utilizadas dentro da IBM no consórcio AGEDIS, mas agora está disponível publicamente.

4.5.3.2 AsmL

Asml (Abstract State Machine Language) [GRS04] é uma linguagem de especificação executável baseada na teoria das máquinas de estado abstratas criada pela Microsoft. A versão atual é AsmL2 (AsnL para Microsoft .NET). Existe uma ferramenta de geração de testes que trabalha com AsmL cujas características estão descritas em [GGSV02]. O algoritmo de geração parece ser baseado na cobertura dastransições da FSM que é obtida a partir da especificação em AsmL. Tal ferramenta não está disponível para venda.

4.5.3.3 PTK

PTK [BBJ+02] é uma ferramenta desenvolvida pelos laboratórios da Motorola do Reino Unido (Motorola Labs UK) para a geração de testes de conformidade a partir de especificações descritas em MSC e PDU(protocol data unit). Ao invés de utilizar FSM para especificar todos os comportamentos possíveis, diagramas MSC são empregados para definir um conjunto de comportamentos aceitos pela IUT. A tradução de MSC para casos de teste para scripts de teste é efetuada tornando explícitas todas as seqüências e dados necessários para executar o comportamento descrito no diagrama, ao invés de escolher um subconjunto de todas as seqüências descritas por uma FSM. A ferramenta PTK interpreta o MSC, fazendo explícitas as seqüências de teste e observações baseadas no comportamento externo que pode ser controlado e aspectos visíveis do MSC, enquanto esconde as ações internas e oculta sua ordem interna. PTK também

provê mensagens de dados, baseadas na PDU (Protocol Data Unit) das especificações, para criar múltiplos casos de teste de um único MSC. As saídas de PTK são conjuntos de teste descritos em TTCN e SDL, mas também possui um gerador de código que pode ser estendido para produzir scripts em outras linguagens.

4.5.4 Ferramentas Acadêmicas

Nas subseções seguintes estão descritas brevemente ferramentas acadêmicas baseadas em modelos de comportamento.

4.5.4.1 MulSaw

MulSaw [MIT06] é um projeto do MIT que incorpora duas ferramentas de geração de testes para programas Java. A primeira ferramenta se chama TestEra [MK01], que aceita entradas especificadas com a linguagem de modelagem Alloy [Jac02]. A segunda ferramenta se chama KORAT [BKM02], que aceita entradas especificadas em JML(Java Modelling Language). JML anotada pré-condições e pós-condições dentro de comentários que ficam no começo dos métodos Java. Os casos de teste gerados por KORAT cobrem todas as instâncias de pré-condição dos métodos, e provê os resultados esperados nas pós-condições.

As ferramentas não estão disponíveis para experimentação.

4.5.4.2 TOSTER

TOSTER (The Object-oriented Software Testing EnviRonment) [oT06] é um sistema de execução e geração de casos de teste produzido pela Warsaw University of Technology. Incorpora tecnologia para mapear a informação dos diagramas UML para o código-fonte de uma aplicação. Também gera e roda os casos de teste baseados nos resultados esperados a partir dos diagramas de estado UML. Parece utilizar duas técnicas de geração, entretanto não utiliza qualquer tipo de diretiva de teste.

4.5.4.3 TorX

TorX [dBRS$^+$00] é uma arquitetura para geração e execução de testes da University of Twente. Dentro dessa arquitetura existe um gerador de testes que aceita propósitos de teste em um formato similar a TGV. As linguagens de modelagem suportadas por TorX são LOTOS, PROMELA [KL] e SDL. O gerador de testes do TorX está incorporado ao pacote de ferramentas CADP [Mat03].

4.5.4.4 TGV

TGV (Test Generation with Verification) [JJ05] é uma ferramenta geradora de casos de teste desenvolvida nos laboratórios do IRISA e VERIMAG. É uma ancestral das ferramentas de geração do projeto AGEDIS. Aceita como entrada especificações em LOTOS, SDL e IF, e produz testes em TTCN. A geração é dirigida via propósitos de teste descritos na forma de IOLTS. TGV tem sido utilizada extensivamente em cenários experimentais e industriais [JJ05],

principalmente na indústria de telecomunicações. O gerador de testes está incorporado dentro do pacote de ferramentas CADP [HJ03] que pode ser obtido a partir da página da WEB.

Capítulo 5

Geração de Casos de Testes em CSP Baseados em Propósitos

Verificação de modelos (*model checking*) [CES86a] e teste (ver Capítulo 2) são duas áreas aparentemente diferentes, pois a primeira lida com análises estáticas de modelos e a segunda com análise dinâmica de implementações. Todavia, estas áreas têm pontos comuns no que diz respeito a geração de seqüências de testes a partir dos contra-exemplos produzidos como resultado de uma verificação de propriedades da especificação.

Verificação de modelos involve o uso de procedimentos de decisão para determinar se uma máquina de estados finita (modelo do sistema), satisfaz uma propriedade Φ dada sob a forma de uma fórmula de lógica temporal (mais informações sobre Lógica Temporal na Seção 4.1.2.1). Se o modelo do sistema satisfaz a propriedade, o resultado da verificação será positivo, no caso contrário, o resultado será negativo. No caso da propriedade ser falsa, ferramentas de verificação de modelos retornam contra-exemplos nas forma de caminhos do sistema que destacam as razões da violação da propriedade.

No contexto onde os testes são extraídos a partir de uma especificação, um propósito de teste pode ser especificado como uma propriedade Φ (geralmente uma fórmula de lógica temporal) da especificação. O que se pretende é extrair traces da especificação onde a propriedade Φ seja válida. Supondo Φ uma propriedade válida da especificação, então, uma alternativa simples para gerar uma seqüência de testes é, fazer a verificação do modelo da especificação contra a negação da propriedade ($\neg\Phi$). A ferramenta de verificação irá provar que $\neg\Phi$ é falso e os contra-exemplos serão essencialmente os traces onde a propriedade Φ é válida [CSE96].

A verificação de refinamentos entre processos CSP (3.3) constitui um tipo específico de verificação de modelos. A ferramenta FDR pode verificar se de acordo com os modelos semânticos de CSP, um processo (implementação) é refinamento do outro (especificação). Considerando o cálculo de refinamento de CSP, processos são ao mesmo tempo utilizados para representar o modelo e as propriedades que se deseja verificar. O comportamento do processo de implementação pode ser visto como uma propriedade válida dentro do modelo semântico do processo de especificação (no caso em que a implementação é um refinamento válido para a especificação). Quando o FDR verifica que um processo (propriedade) não é refinamento de outro (modelo), são retornados contra-exemplos que expõem a violação das propriedades requeridas no modelo semântico do refinamento.

Partindo deste princípio, podemos definir um processo CSP de propósito de testes que especifica propriedades para as quais se pretende gerar testes e utilizar este processo de tal forma que, através de expressões de refinamento, seja possível obter cenários de teste (traces de contra-exemplo) onde as propriedades especificadas estão presentes. Propósitos de teste

CSP podem ser utilizados para descrever propriedades *safety* (sobre a eventual ocorrência de eventos no comportamento da especificação) a partir das quais pode-se extrair cenários de teste da especificação. Fica assim estabelecido o princípio da geração de testes CSP a partir de propósitos de teste CSP.

Ao longo deste capítulo é mostrada uma abordagem de geração de casos de teste coerente (*sound*) dirigida por propósitos de teste e uma teoria de testes, ambas formuladas exclusivamente com operadores e ferramentas de CSP. Verificações de refinamento entre processos e os contra-exemplos retornados serão as ferramentas fundamentais para o desenvolvimento desta abordagem.

5.1 Especificação de Exemplo

Para ilustrar a abordagem, será utilizada a seguinte especificação de exemplo. Este exemplo é puramente simbólico e baseado nas especificações apresentadas em [JJ05], pois mostrou-se útil para exercitar vários aspectos da geração de testes.

01	$channel\, a, b, c, y, z$
02	$channel\, t$
03	
04	$AlfaSi = \{a, b, c\}$
05	$AlfaSo = \{y, z\}$
06	$AlfaS = AlfaSi \cup AlfaSo$
07	
08	$S0 = t \rightarrow S9 \,\Box\, t \rightarrow S2$
09	$S2 = t \rightarrow S0 \,\Box\, (c \rightarrow S6 \,\Box\, b \rightarrow S4)$
10	$S4 = z \rightarrow S2 \,\Box\, t \rightarrow S4 \,\Box\, t \rightarrow S8$
11	$S6 = y \rightarrow S7$
12	$S7 = c \rightarrow S6$
13	$S8 = y \rightarrow S0$
14	$S9 = a \rightarrow S8$
15	
16	$SYSTEM = S0 \setminus \{t\}$

No exemplo, o processo *SYSTEM* modela o comportamento arbitrário de um sistema não-determinístico. O alfabeto da especificação *AlfaS* ou Σ_{SYSTEM} (linha 6) resulta da união de dois conjuntos de eventos, $\Sigma_{SYSTEMi}$ ou $AlfaSi = \{a, b, c\}$ (linha 4) que corresponde aos eventos de entrada, e $\Sigma_{SYSTEMo}$ ou $AlfaSo = \{y, z\}$ (linha 5) que corresponde aos eventos de saída. O evento *t* (linhas 8, 9 e 10) é utilizado para especificar eventos internos do sistema; é declarado como $channel\, t$ (linha 2). *t* torna-se um evento interno pela aplicação da operação de hiding sobre a equação de *S*0: $S0 \setminus \{t\}$ (linha 16). Eventos internos serão utilizados para criar situações de não-determinismo que podem trazer impacto durante a análise da seleção dos testes.

Processos como *S*0, *S*2, *S*4, *S*6, *S*7, *S*8 e *S*9 (linhas 8 a 14), são utilizados por *SYSTEM* (linha 16) para modelar estados do sistema que possuem não-determinismo e divergência.

Um exemplo de situação de não-determinismo encontra-se no comportamento do processo $S0$: que de forma não-determinística pode se comportar como $S9$ ou $S2$, pois a escolha externa $(t \rightarrow S9 \,\square\, t \rightarrow S2) \setminus \{t\}$ tem comportamento equivalente a $S9 \sqcap S2$ (este último trivialmente não-determinístico). Como exemplo de situação de divergência, nota-se o processo $S2$, que pode se comportar de acordo o lado esquerdo de sua escolha externa: $t \rightarrow S0$. Tal escolha leva o processo $S2$, de forma não visível e sem influência do ambiente, a comportar-se como $S0$. $S0$ por sua vez, pode comportar-se de acordo com o lado direito de sua escolha externa: $t \rightarrow S2$. Esta escolha leva o processo $S0$, também através de um evento interno, a comportar-se como $S2$. Este fluxo de ações do sistema leva a uma divergência.

O motivo para o alfabeto da especificação estar dividido em entradas e saídas, é que a abordagem proposta distingue entradas e saídas, o que facilitará a relação dessa abordagem de geração com outras abordagens que se baseiam na relação de conformidade **ioco** (ver Seção 4.4.1.4). Desta maneira, a abordagem de geração pode ser aplicada inclusive para sistemas reativos [Bri06], que recebem um estímulo de entrada e reagem produzindo uma resposta.

Apesar de trabalhar na semântica denotacional de CSP, na Figura 5.1 está ilustrada a semântica operacional do processo *SYSTEM*, que será utilizada para facilitar o raciocínio com relação as operações que serão realizadas durante a seleção e geração de testes.

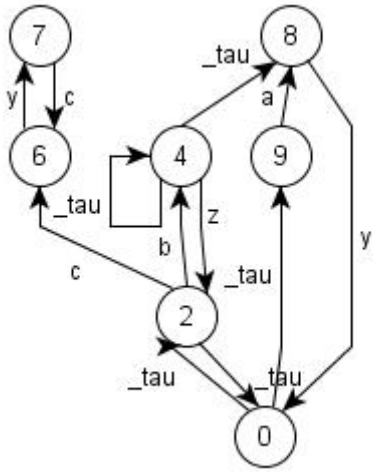

Figura 5.1 LTS do processo SYSTEM

Cada processo da especificação possui um estado correspondente ao LTS da Figura 5.1: $S0$ corresponde ao estado 0, $S2$ ao estado 2; e assim por diante, para os processos $S4$, $S6$, $S7$, $S8$ e $S9$.

Na seção seguinte é revisada e adaptada uma técnica de validação de cenários (seqüências de eventos de uma especificação) que servirá de ponto de partida para entender como se processa a seleção dos testes por propósitos de teste na Seção 5.3.

Todas os exemplos de especificação CSP apresentados neste capítulo estão listados na versão de CSP lida por ferramentas (CSP_M) no Apêndice A.

5.2 Processo *MATCHRW*(*seq*)

Antes de detalhar a abordagem de seleção e geração de casos de teste, está seção apresenta uma releitura sobre o trabalho de Rasch e Wehrheim em [RW05], que propõe um método para verificar a validade de cenários em modelos CSP através da verificações de refinamentos de processos. Os conceitos fundamentais deste método são estendidos nas seções seguintes onde é detalhada a abordagem de seleção com propósitos de teste CSP.

Considerando o comportamento modelado como um processo P com alfabeto de eventos Σ_P, um cenário é uma seqüência de eventos *seq* cujos eventos pertencem ao alfabeto Σ_P. É possível verificar se um cenário *seq* é ou não válido num processo P pelos seguintes predicados.

1. $\exists\, t : runs(P);\ u : Seq\Sigma_P;\ v : Seq\Sigma_P \bullet t = u \frown seq \frown v$
2. $\neg \exists\, t : runs(P);\ u : Seq\Sigma_P;\ v : Seq\Sigma_P \bullet t = u \frown seq \frown v$

Na primeira opção, é verificado se *seq* ocorre eventualmente em pelo menor um dentre os possíveis traces t produzidos pelo processo P, onde $runs(P)$ denota o conjunto de todos os traces produzidos por P. Isto é, o predicado 1 verifica se existe algum trace $t : runs(P)$ com prefixo u e sufixo v tal que $t = u \frown seq \frown v$ (onde $\frown$ é o operador de concatenação de seqüências). De forma complementar, a segunda verificação é a negação na primeira: verifica que não existe um trace $t : runs(P)$ com prefixo u e sufixo v tal que $t = u \frown seq \frown v$. [RW05] define um refinamento de processos CSP cujo resultado positivo significa que a primeira proposição é válida, e no caso negativo que a segunda proposição é verdadeira. A caracterização de tal refinamento utiliza um processo *MATCHRW*(*seq*) que tem a habilidade de detectar a ocorrência de *seq* (cenário) dentro dos possíveis eventos sincronizados entre o processo e o ambiente. A definição de *MATCHRW*(*seq*) é dada pela especificação:

01	$MATCHRW(seq)$	$=$	$MATCHRW'(\Sigma_P, seq)$
02			
03	$MATCHRW'(\Sigma, \langle\rangle)$	$=$	$match \rightarrow Stop$
04	$MATCHRW'(\Sigma, s)$	$=$	$\text{head}(s) \rightarrow MATCHRW'(\Sigma, \text{tail}(s))$
05			$\Box$
06			$(\Box\, ev : \Sigma - \{\text{head}(s)\} @ ev \rightarrow MATCHRW(seq))$

MATCHRW(*seq*) se comporta como o processo parametrizado $MATCHRW'(\Sigma, s)$ (linha 4) quando este último recebe como valores o alfabeto de eventos de P (Σ_P) e a seqüência de eventos (*seq*) que se pretende verificar a ocorrência em $runs(P)$. O processo $MATCHRW'(\Sigma_P, seq)$ (linha 1) funciona como uma máquina que busca a ocorrência dos eventos de *seq* dentro dos eventos oferecidos pelo ambiente em que se encontra. A pesquisa acontece evento a evento com o qual o processo sincroniza, tentando encaixar os eventos encontrados na ordem do primeiro ao último dos eventos de *seq*. $MATCHRW'(\Sigma_P, seq)$ considera que os eventos sincronizados são oferecidos arbitrariamente e pertencem ao conjunto Σ_P. Caso o evento sincronizado não corresponde ao evento esperado em *seq* (linha 6), a busca é reiniciada (especificação se comporta como *MATCHRW*(*seq*) na linha 3). Quando a ordem dos eventos encontrados retrata um prefixo de *seq* (linha 4), a busca continua a partir dos eventos consecutivos ($MATCHRW'(\Sigma, \text{tail}(s))$

na linha 4). O fim da busca acontece quando são sincronizados todos os eventos de *seq*, neste ponto, o processo comunica o evento *match* (onde $match \notin \Sigma_P$) e se comporta como *Stop* (linha 1).

Abaixo consta uma versão parametrizada do refinamento proposto em [RW05].

$$(P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)) \setminus \Sigma_P \sqsubseteq_\tau match \rightarrow Stop \tag{5.1}$$

Analisando o refinamento (5.1), destaca-se como diferença com relação ao original em [RW05] a utilização de um processo parametrizado *MATCHRW*(*seq*). No refinamento em [RW05] não existe parametrização, sendo o referido processo montado de acordo com o valor de *seq*. Tal diferença será importante para acompanhar a extensão da idéia apresentada na Seção 5.3, onde serão apresentados os processos especiais utilizados para seleção de traces de teste.

Na relação de refinamento (5.1), *MATCHRW*(*seq*) tem como ambiente os eventos comunicados por *P*, pois *MATCHRW*(*seq*) é colocado em paralelo com *P* para sincronizar as ações de acordo com o conjunto de eventos Σ_P. A progressão do paralelismo $P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)$ inicia e acontece livremente, até que ocorra *seq* como parte do trace do paralelismo; neste momento, o evento *match* é comunicado e *P* fica impedido de progredir, pois *match* está fora do alfabeto de sincronização Σ_P. Se a comunicação de *match* acontece, naturalmente este evento pode ser encontrado nos traces do paralelismo.

A verificação do cenário *seq* é formulada como uma relação de refinamento em (5.1). Assumindo que *seq* é um cenário válido de *P*, então tal refinamento precisa identificar nos traces de $P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)$ a existência de traces marcados com o evento *match*. Portanto, sendo *match* o evento de interesse a ser analisado emprega-se a operação $\setminus\,\Sigma_P$ para que todos os eventos de Σ_P comunicados por *P* tornam-se eventos internos (invisíveis na semântica de *Traces*). Apenas *match* permanece visível. Encontrar o evento *match* nos traces de $(P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)) \setminus \Sigma_P)$ significa que existe um ou mais pontos de *P* onde o cenário *seq* pôde ser encontrado, portanto:

$$traces((P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)) \setminus \Sigma_P)) \subseteq \{\langle\rangle, \langle match\rangle\} \tag{5.2}$$

O processo $match \rightarrow Stop$ possui o seguinte conjunto de traces $\{\langle\rangle, \langle match\rangle\}$, que corresponde ao lado direito da relação (5.2). Por conseguinte, a verificação de (5.2) pode ser reformulada em termos dos traces dos processos $(P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)) \setminus \Sigma_P)$ e $match \rightarrow Stop$:

$$traces((P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)) \setminus \Sigma_P)) \subseteq traces(match \rightarrow Stop) \tag{5.3}$$

De acordo com a semântica de *Traces* de CSP (ver Seção 3.3) a verificação da relação (5.3) se resume à expresão (5.1). Exemplificando o funcionamento de *MATCHRW*(*seq*), pode-se verificar no processo *SYSTEM* (da seção 5.1) a validade de dois cenários: $MATCHRW(\langle a, y\rangle)$ e $MATCHRW(\langle a, b\rangle)$. Para o primeiro cenário, tem-se a expressão de refinamento:

$$(SYSTEM\,|[\,\Sigma_P\,]|\,MATCHRW(\langle a,y\rangle)) \setminus \Sigma_{SYSTEM} \sqsubseteq_\tau match \rightarrow Stop \tag{5.4}$$

O resultado do refinamento (5.4) é positivo, o que indica que existe ao menos um trace da especificação que contém o cenário $\langle a, y\rangle$. Isto pode ser confirmado observando a representação

operacional de *SYSTEM* na Figura 5.1, onde pode ser encontrado o trace $\langle a,y\rangle$ cujo sufixo é *seq*. Para a validação do segundo cenário tem-se o seguinte refinamento:

$$(SYSTEM\,|[\,\Sigma_P\,]|\,MATCHRW(\langle a,b\rangle)) \setminus \Sigma_{SYSTEM} \sqsubseteq_\tau match \rightarrow Stop \tag{5.5}$$

O resultado do refinamento (5.4) é negativo; isto demonstra que não há trace *t* do sistema especificado tal que $\langle a,b\rangle$ seja sufixo de *t*.

5.3 Seleção de cenários de teste

A seção anterior apresenta uma adaptação do processo *MATCHRW*(*seq*) proposto em [RW05]. Este processo é utilizado em uma relação de refinamento para informar se um dado cenário é válido ou não, com respeito a uma especificação CSP. Tal refinamento, no caso de validação positiva (quando o cenário é identificado na especificação), não é capaz de explicitar exemplos de traces nos quais o cenário verificado ocorre.

Considerando *seq* com uma propriedade da IUT que se pretende verificar, partindo de uma especificação *P*, pretende-se conhecer os traces onde a propriedade fornecida é válida. Estes traces serão chamados de cenários de teste (diferente de caso de teste da Seção 4.4). Um conjunto de cenários de teste pode ser entendido como um conjunto (finito ou infinito) de traces da especificação *P* que possuem *seq*. São traces que levam o observador do sistema (ex: o testador) a alcançar a propriedade *seq* dentro das infinitas possibilidades de traces existentes. Cada cenário é uma seqüência de eventos da especificação que partem do estado inicial do sistema e alcança os pontos da especificação onde ocorre *seq*. Esta seção apresentada uma técnica de obtenção de cenários de teste.

A relação de refinamento (5.4) revela que o trace $\langle a,y\rangle$ é um cenário válido da especificação *SYSTEM*. No entanto, surge outra questão: como obter o(s) trace(s) de *SYSTEM* onde $\langle a,y\rangle$ pode ser encontrado? Considerando que um mesmo cenário pode ser repetido infinitas vezes dentro de uma especificação, em primeira instância, pode-se almejar obter o menor caminho que o alcança. Na Figura 5.2, pode-se visualizar a semântica operacional da expressão $SYSTEM\,|[\,\Sigma_P\,]|\,MATCHRW(\langle a,y\rangle)$ do refinamento (5.4).

Na Figura 5.2, seguindo as transições que passam pelos estados root, 1, 3, 6, 10 e 12, verifica-se facilmente que o cenário $\langle a,y\rangle$ está presente no comportamento do processo *SYSTEM*. O caminho formado por estas transições, $\langle a,y,match\rangle$, consiste no menor trace que leva do estado inicial do sistema até o evento *match*. Um simples algoritmo de menor caminho pode ser usado para percorrer o grafo e encontrar o menor trace, porém o nosso objetivo aqui é formular esta busca como um problema de refinamento entre processos.

Uma ferramenta de verificação de refinamentos como FDR [Sys05] pode fornecer o menor caminho até um evento da especificação, bastando para isto formular o problema em termos de refinamentos entre processos CSP. O algoritmo de verificação percorre o comportamento dos processos na profundidade (*deep nodes first* — DNF) e, na situação em que o refinamento não e válido, retorna o menor caminho para o primeiro contra-exemplo encontrado.

Um refinamento de traces pode retornar como contra-exemplo o trace onde ocorre um evento específico, basta que o evento esteja no lado direito e não no lado esquerdo do refina-

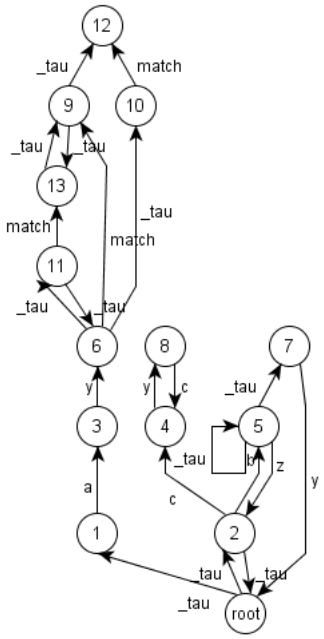

Figura 5.2 LTS da verificação do cenário $\langle a, y \rangle$

mento. O evento em questão é o *match*, que é utilizado para marcar os pontos da especificação onde ocorre um cenário de validação *seq*.

Para entender como, através de refinamentos e contra-exemplos, obter os cenários onde ocorre $seq^\frown match$ é necessário raciocinar sobre o refinamento de traces entre dois processos. O refinamento de traces entre dois processos idênticos $P \sqsubseteq_\tau P$ é trivialmente verdadeiro pois $traces(P) \supseteq traces(P)$. No entanto, uma vez que o processo original P (especificação) de alfabeto Σ_P é modificado e se torna um processo P' (implementação) com alfabeto $\Sigma_{P'}$, pelo acréscimo de um evento *ev* tal que $ev \in \Sigma_{P'}$ e $ev \notin \Sigma_P$, logicamente:

$$traces(P) \not\supseteq traces(P') \Rightarrow P \not\sqsubseteq_\tau P'$$

Este resultado ocorre porque:

$$\forall\, t^\frown \langle ev \rangle \in traces(P') \bullet t \in traces(P) \wedge ev \in \Sigma_{P'} \wedge ev \notin \Sigma_P \Rightarrow traces(P) \not\supseteq traces(P')$$

Considerando a proposição acima, embora existam traces *t* que estão presentes em *P* quanto em P', $t^\frown \langle ev \rangle$ está presente apenas em P', pois *P* não contém *ev* em seu alfabeto nem em seus traces. $t^\frown \langle ev \rangle$ faz parte dos contra-exemplos do refinamento $P \sqsubseteq_\tau P'$, pois é um trace que está presente na implementação e não está na especificação. Tendo em mente estas características do refinamento de traces, e conhecendo o comportamento do paralelismo $P|[\Sigma_P]|MATCHRW(seq)$ chega-se ao seguinte refinamento que retorna como contra-exemplo o menor trace com sufixo $seq^\frown < match >$:

$$P \sqsubseteq_\tau P|[\Sigma_P]|MATCHRW(seq) \tag{5.6}$$

Raciocinando na proposição anterior, quando P' é substituído por $P|[\Sigma_P]|MATCHRW(seq)$, t por $t^\frown seq$ e ev por $match$, o resultado da verificação do refinamento

$$P \sqsubseteq_\tau P|[\Sigma_P]|MATCHRW(seq)$$

é falso. Os contra-exemplos (ou cenário de teste) encontrados são traces da forma $t^\frown seq^\frown\langle match\rangle$.

Comparando a Figura 5.1 que correponde a P, e a Figura 5.2 que corresponde a P', pode-se observar que um dos contra-exemplos do refinamento

$$SYSTEM \sqsubseteq_\tau SYSTEM|[\Sigma_P]|MATCHRW(\langle a,y\rangle)$$

é igual a $\langle a,y,match\rangle$. Este cenário de teste corresponde ao menor caminho para alcançar $\langle a,y\rangle$, e segue a forma $t^\frown seq^\frown\langle match\rangle$ onde, $t = \langle\rangle$ e $seq = \langle a,y\rangle$.

No caso em que o cenário não pode ser encontrado na especificação do sistema, nenhum cenário de teste é obtido pelo refinamento. Neste caso, os comportamentos de

$$P|[\Sigma_P]|MATCHRW(seq)$$

e P são equivalentes pelo modelo de *Traces*, isto é, o resultados das verificações abaixo são positivos.

$$P|[\Sigma_P]|MATCHRW(seq) \sqsubseteq_\tau P$$
$$P \sqsubseteq_\tau P|[\Sigma_P]|MATCHRW(seq)$$

O refinamento (5.6) retorna apenas um cenário de teste, contando que FDR está configurado na opção padrão que retorna apenas um resultado para refinamentos. Para obter mais de um cenário é necessário aplicar uma extensão desta estratégia.

5.3.1 Derivação de Múltiplos Cenários

Um observador do sistema pode optar pela exploração de cenários de teste alternativos, onde as seqüências de ações que antecedem *seq* são diversas do menor caminho. Esta subseção descreve uma extensão do uso do refinamento onde o menor cenário de teste obtido $t^\frown seq$ é o ponto de partida para obtenção de outros cenários $u^\frown seq$, tal que $u \neq t$.

FDR retorna (como opção padrão), de acordo com o modelo semântico utilizado, o contra-exemplo mais simples quando o resultado da verificação do refinamento entre dois processos é falso. O modelo em questão é o de *Traces* onde: o menor trace extra, encontrado no processo do lado direito do refinamento (5.6), é utilizado como contra-exemplo, pois este trace corresponde a um comportamento adicional inexistente no processo do lado esquerdo. O contra-exemplo retornado por (5.6) é o trace $t^\frown seq^\frown\langle match\rangle$ que será denotado como CE_1. Uma forma de fazer com que FDR retorne um contra-exemplo CE_2 diferente de CE_1, é incorporar CE_1 como parte do comportamento do processo do lado esquerdo do refinamento. A forma mais simples é construir um processo, a ser incorporado como comportamento da especificação, que comunica cada evento de CE_1 ao mesmo tempo que preserva a ordem dos eventos. Abaixo está a definição de tal processo:

$PREFIX(\langle\rangle) = Stop$
$PREFIX(seq) = \text{head}(seq) \rightarrow PREFIX(\text{tail}(seq))$

O processo *PREFIX* recebe como parâmetro o trace dos contra-exemplos encontrados e gera um processo cujo comportamento é este trace. No contexto onde CE_1 é o único contra-exemplo conhecido, $PREFIX(CE_1)$ é combinado ao comportamento de P através do operador de escolha externa. CE_2 é obtido a partir do refinamento:

$$P \,\Box\, PREFIX(CE_1) \sqsubseteq_\tau P\,|[\,\Sigma_P\,]|\,MATCHRW(seq)$$

Para a semântica de *Traces*, o comportamento do lado esquerdo de P escolha externa $PREFIX(CE_1)$ equivale a estender o comportamento de P pelo acréscimo do trace CE_1. Desde que CE_1 é um possível comportamento do lado esquerdo (e também do lado direito), o resultado da verificação do refinamento quando falso retorna um contra-exemplo $CE_2 \neq CE_1$. Considerando CE_1 da forma $t \frown seq \frown \langle match\rangle$ e CE_2 da forma $u \frown seq \frown \langle match\rangle$, então $u \neq t$. A aplicação incremental deste refinamento consiste em aproveitar os contra-exemplos já obtidos $\{CE_1, CE_2, ..., CE_N\}$ para obter um próximo diferente CE_{N+1} (caso exista). A forma geral para obter o contra-exemplo CE_{N+1} a partir dos anteriores $\{CE_1, CE_2, ..., CE_N\}$ é:

$$P \,\Box\, PREFIX(CE_1) \,\Box\, PREFIX(CE_2) ... \,\Box\, PREFIX(CE_N) \sqsubseteq_\tau P\,|[\,\Sigma_P\,]|\,MATCHRW(seq) \quad (5.7)$$

Como os contra-exemplos retornados pelo FDR são sempre os mais curtos, a aplicação incremental do refinamento (5.7) retorna inicialmente os traces menores. A medida que os traces são incorporados ao comportamento da especificação P, e verificado novamente o refinamento, os próximos contra-exemplos obtidos possuem um comprimento no mínimo igual aos já incorporados. Quando estão esgotados os contra-exemplos em dada profundidade, imediatamente são retornado outros de maior comprimento.

Quando a quantidade de traces relacionados à propriedade *seq* é finita, e todos os contra-exemplos possíveis já foram obtidos pela aplicação incremental de N vezes o refinamento (5.7), o resultado da verificação será positivo e nenhum contra-exemplo CE_{N+1} será retornado por FDR.

Como visto, FDR retorna da especificação *SYSTEM* o cenário $CE_1 = \langle a, y, match\rangle$, quando é verificado o refinamento (5.6). Utilizando o contra-exemplo CE_1 pode-se montar uma nova expressão de refinamento para que o FDR identifique CE_2.

$$SYSTEM \,\Box\, PREFIX(CE_1) \sqsubseteq_\tau SYSTEM\,|[\,\Sigma_P\,]|\,MATCHRW(\langle a, y\rangle)$$

O resultado da verificação acima leva FDR a identificar $CE_2 = \langle b, y, a, y, match\rangle$. Aproveitando CE_1 e CE_2, pode-se montar o refinamento que identifica CE_3.

$$SYSTEM \,\Box\, PREFIX(CE_1) \,\Box\, PREFIX(CE_2) \sqsubseteq_\tau SYSTEM\,|[\,\Sigma_P\,]|\,MATCHRW(\langle a, y\rangle)$$

O resultado da verificação acima leva FDR a identificar $CE_3 = \langle b, z, a, y, match\rangle$. Aproveitando CE_1 ,CE_2 e CE_3 pode-se montar o refinamento que identifica CE_4.

$$SYSTEM \,\Box\, PREFIX(CE_1) \,\Box\, PREFIX(CE_2) \,\Box\, PREFIX(CE_3) \sqsubseteq_\tau SYSTEM \,|[\,\Sigma_P\,]|\, MATCHRW(\langle a, y\rangle)$$

O resultado da verificação acima leva FDR a identificar $CE_4 = \langle b, z, b, y, a, y, match\rangle$.

Quando a quantidade existente de cenários de teste que levam a *seq* é infinita, podem ser obtidos quantos cenários de teste quanto se queira para a mesma especificação. No caso do processo *SYSTEM*, existem infinitas possibilidades de cenários que podem ser encontrados pela aplicação incremental do refinamento, quando a propriedade em questão é $\langle a, y\rangle$. A Figura 5.2 mostra que existem infinitos contra-exemplos da forma $u \frown \langle a, y\rangle \frown \langle match\rangle$ de prefixo u em *SYSTEM*. Os contra-exemplos CE_2, CE_3 e CE_4 são diferenciados pelos prefixos $\langle b, y\rangle$, $\langle b, z\rangle$ e $\langle b, z, b, y\rangle$ que levam a propriedade $\langle a, y\rangle$.

5.4 Seleção de cenário via Propósitos de Teste CSP

Na Seção 4.1.2.1, foi apresentado o conceito de propósitos de testes e os vários formalismos utilizados para descrevê-los. Como visto, a literatura utiliza CSP apenas como uma notação para especificação, porém, aqui, é empregado também para especificar propósitos de teste. Na Seção 5.3 é apresentada uma técnica de extração de testes baseadas em refinamentos de processos para obter cenários de teste a partir de uma propriedade *seq* fornecida. O processo *MATCHRW*(*seq*) é projetado de forma que a expansão do paralelismo dele com o processo de especificação *P* demarca os cenários de *P* onde eventualmente ocorre *seq*. Este processo encapsula a ideia fundamental de um propósito de teste CSP. Esta seção amplia a visão com respeito às possibilidades de como se especificar cenários de teste utilizando propositos de teste CSP com o apoio das técnicas de verificação de refinamento utilizadas na seção anterior.

O propósito de teste CSP (PTCSP) segue a definição de [LdBB+01]: são especificações parciais dos casos de teste obtidos. É constituído de um ou vários processos que especificam cenários para um processo *P* do qual pretende-se extrair testes. Operacionalmente, isto pode ser obtido quando se coloca o processo de especificação *P* em paralelo com o processo de propósito de testes *PT*, e aplica-se algumas verificações de refinamentos que retornam os cenários de teste selecinados (de acordo com a expressão de refinamento (5.6)). O termo **produto paralelo** será utilizado para indicar o processo que resulta da composição paralela entre uma especificação *P* e um propósito de teste CSP, conforme a expressão abaixo.

$$PARALLEL\_PRODUCT = P \,|[\,\Sigma_P\,]|\, PTCSP \tag{5.8}$$

Seja Σ_P o alfabeto de eventos da especificação representada pelo processo *P*, e Σ_{PTCSP} o alfabeto de eventos do processo do propósito de testes. Qualquer processo *PTCSP* que obdedeça os seguintes requisitos é considerado um propósitos de teste CSP.

- *PTCSP* possui comportamento determinístico;
- Seja $\Sigma_{mark} = \{|\, accept \,|\} \cup \{|\, refuse \,|\}$ o alfabeto de eventos demarcadores, tal que $\Sigma_P \cap \Sigma_{mark} = \emptyset$, então $\Sigma_{PTCSP} \subseteq \mathbb{P}(\Sigma_P \cup \Sigma_{mark})$; e,

- Todos os contra-exemplos retornados pela verificação de refinamento da expressão

  $$P \sqsubseteq_\tau PARALLEL\_PRODUCT$$

  são da forma $t^\frown\langle mark\rangle$, onde $t \in traces(P)$ e $mark \in \Sigma_{mark}$, e t **conforms** [LdBB+01] a especificação P, isto é, $P \sqsubseteq_\tau t$.

São infinitas as possibilidades de construção de processos de acordo com as definições acima. A Figura 5.3 lista um conjunto de processos composicionais apropriados para capturar várias operações de seleção de cenários que se pode realizar partindo dos refinamentos descritos na Seção anterior. Os processos *ACCEPT*, *REFUSE*, *ANY*, *NOT* (linhas 5 a 12) são as formas mais simples (ou primitivas) de definir propósitos de teste em CSP. Já os processos *MATCH*, *EXCEPT* e *UNTIL* (linhas 14 a 18) são obtidos pela combinação dos anteriores e constituem operadores de maior expressividade. As subseções seguintes explicam em detalhes o comportamento de cada um dos processos da Figura 5.3.

01	$MAX\_MARKS = 10$
02	
03	$channel\, accept, refuse : 1..MAX\_MARKS$
04	
05	$ACCEPT(\{id\}) = accept.id \rightarrow Stop$
06	
07	$REFUSE(\{id\}) = refuse.id \rightarrow Stop$
08	
09	$ANY(evset, next) = \Box\ ev : evset @ ev \rightarrow next$
10	
12	$NOT(\Sigma_P, evset, next) = ANY(\Sigma_P - evset, next)$
13	
14	$MATCH(\Sigma_P, evset, next, initial) = ANY(evset, next) \,\Box\, NOT(\Sigma_P, evset, initial)$
15	
16	$EXCEPT(\Sigma_P, evset, next, initial) = MATCH(\Sigma_P, evset, initial, next)$
17	
18	$UNTIL(\Sigma_P, evset, next) = RUN(\Sigma_P - evset) \,\triangle\, ANY(evset, next)$

Figura 5.3 Processos básicos para PTCSP

5.4.1 Processos *ANY* e *NOT*

Os processos $ANY(evset, next)$ e $NOT(\Sigma, evset, next)$ são os construtores fundamentais para criar propósitos de teste (linhas 8 e 9 da Figura 5.3). O primeiro é uma escolha externa dos eventos $ev \in evset$, onde $evset$ é necessariamente um subconjunto de eventos do alfabeto de P ($evset \subseteq \Sigma_P$). Quando posto no lado direito da expressão (5.8), sincroniza com qualquer um

dos eventos oferecidos por *P* que estejam em *evset* e, em seguida o processo *ANY*, se comporta como o processo de continuação *next*. A sincronização do evento *ev* com um evento oferecido pela especificação *P* equivale a uma busca com sucesso por este mesmo evento dentro dos traces de *P*. Se ao invés de um evento, se pretende buscar por toda uma seqüência (*seq*) de eventos de *P*, utiliza-se a composição:

$$
\begin{array}{lcl}
ANY_{seq} & = & ANY(element(1, seq), ANY_2) \\
ANY_2 & = & ANY(element(2, seq), ANY_3) \\
... & & \\
ANY_{SEQ\_SIZE} & = & ANY(element(SEQ\_SIZE, seq), match \rightarrow Stop)
\end{array}
$$

Figura 5.4 Reconhecimento de seqüência com *ANY*

Na especificação acima, seja *seq* uma seqüência de tamanho *SEQ_SIZE* com eventos *ev* tal que $ev \in \Sigma_P$, e *element* uma função $INT \times SEQ \longmapsto \Sigma_P$, que recebe como argumentos um inteiro (*INT*) que representa a posição de um elemento na seqüência, e uma seqüência de eventos (*SEQ*) para retornar o evento da seqüência como na posição dada. O comportamento de ANY_{seq} é sincronizar com cada elemento $element(n, seq)$, de $n = 1$ até $n = SEQ\_SIZE$. Se colocado no lado direito da expressão (5.8) não se comporta ainda como $MATCHRW(seq)$, porque ANY_{seq} sincroniza até o último evento de *seq* ($element(SEQ\_SIZE, seq)$, se e somente se, for um prefixo de *P* ($seq \in traces(P)$). Caso um dos processos ANY_n não sincronize com o evento esperado em *P*, ocorre um deadlock e o processo de seleção não segue adiante. Já $MATCHRW(seq)$ procura pela eventual ocorrência de *seq*: considera que antes de ocorrer a seqüência podem acontecer vários outros eventos intermediários, até que haja a sincronização de todos os elementos.

O processo $NOT(\Sigma, evset, next)$ tem comportamento complementar a $ANY(evset, next)$, portanto o primeiro é definido a partir do segundo. Apesar de ter comportamento equivalente a $ANY(\Sigma - evset, next)$ (linha 10 da Figura 5.3), *NOT* cria mais facilidade para expressar PTCSPs, pois constitui uma forma mais sucinta do que sua própria definição. Fica como escolha do Projetista de Testes saber quando é mais conveniente utilizá-lo.

Apesar do evento *match* ter sido utilizado para ilustrar o comportamento deste dois processos básicos, ele não faz parte da notação de eventos utilizados pelos propósitos de teste CSP; *accept* e *refuse* são utilizados em sua substituição. A próxima subseção trata da utilização destes últimos eventos.

5.4.2 Processos *ACCEPT*, *REFUSE*

Os processos $ACCEPT(\{id\})$ e $REFUSE(\{id\})$ (linhas 4 e 5 da Figura 5.3) são empregados nas extremidades dos propósitos de teste, isto é, como valor do parâmetro *next*. São utilizados para demarcar o comportamento do produto paralelo quando ocorre a expansão do paralelismo (5.8). Estes processos comunicam eventos *accept.id* e *refuse.id* que barram a expansão do produto paralelo, logo que finda a seleção de um cenário de testes. Apesar destes processos gozarem de comportamentos semelhantes, as marcação produzidas por cada um deles é utilizada de forma distinta. *ACCEPT* marca pontos de interesse da especificação, posteriormente

utilizados para extrair cenários de teste (evento *accept*.*id*). *REFUSE* marca cenários a serem excluídos (não relevantes) da seleção (evento *refuse*.*id*). Os cenários marcados com *refuse*.*id* não são utilizados para derivar casos de teste; a utilidade básica é economizar esforço computacional pelo impedimento da expansão do paralelismo logo que identificado um cenário que não seja de interesse. O valor comunicado em *id* está restrito ao domínio $id \in \{0...MAX\_MARKS\}$, é passado como parâmetro (um conjunto unitário $\{id\}$) em ambos os processos. O valor definido para *MAX_MARKS* (linha 1 na Figura 5.3) foi arbitrário, na prática, deve ser ajustado de acordo com a necessidade de utilizar um conjunto maior ou menor de valores disponível para marcação.

Um PTCSP pode demarcar diferentes cenários de seleção: cada especificação de cenário se diferencia pelo uso de processos *ACCEPT*($\{id\}$) ou *REFUSE*($\{id\}$) como último comportamento do processo. Por exemplo, considere um propósito de testes que selecione dois cenários simultâneos a partir do processo *SYSTEM*:

Cenário 1 Eventualmente acontece o evento *y* e, depois, eventualmente acontece o evento *z*. Este cenário é marcado com *accept*.1 (que é comunicado pelo processo *ACCEPT*($\{1\}$)); e,

Cenário 2 Eventualmente ocorre *z*. Este cenário é marcado com *refuse*.1 (que é comunicado pelo processo *REFUSE*($\{1\}$)).

Na Figura 5.5 está uma das possíveis formas de especificar um PTCSP que captura os cenários descritos informalmente acima. O cenário 1 está especificado nas linhas 1,7,8 e 9. Na linha 1, espera-se que eventualmente seja oferecido, pela especificação (*SYSTEM*) na expansão do produto paralelo, o evento *y* e em seqüência se comporta como o processo PT' (linha 7). PT' aguarda que a especificação eventualmente ofereca *z* e sincroniza (linha 7) com este evento, em sequida, marca o produto paralelo com o evento *accept*.1 (*ACCEPT*($\{1\}$)). Caso contrário, se *z* não é oferecido, PT' sincroniza com qualquer outro evento oferecido do alfabeto da especificação que seja diferente de *z* (linha 9), e volta a esperar pela eventual ocorrência de *z* comporta-se como PT'. O cenário 2 está especificado na linha 3. Neste ponto, o processo espera que o evento *z* seja oferecido (quando *y* não foi comunicado antes) e sincroniza para demarcar o produto paralelo com o evento *refuse*.1 (*REFUSE*($\{1\}$). Para qualquer outros cenários diferentes dos descritos, quando os eventos oferescidos são diferentes de *y* e *z*, o processo sincroniza e volta a se comportar como *PT* (linha 5).

O processo *PT* é um exemplo de propósito de teste CSP válido de acordo com as propriedades requeridas para um PTCSP:

- *PT* possui comportamento determinístico. Aplicando a verificação de determinismo de processos de FDR chega-se ao resultado que *PT* é determinístico. Um processo não determinístico pode levar a seleção inconsistente de cenários de teste, conseqüentemente é evitado;

- Em PT, o conjunto de eventos envolvidos na definição dos cenários ($\{y,z\}$) é subconjunto do conjunto de eventos do alfabeto da especificação (*SYSTEM*), isto é $\{y,z\} \subseteq \Sigma_{SYSTEM}$: o que pode ser facilmente notado pela comparação dos dois conjuntos; e,

01	PT	$=$	$ANY(y, PT')$
02			$\Box$
03			$ANY(\{z\}, REFUSE(\{1\}))$
04			$\Box$
05			$NOT(\Sigma_{SYSTEM}, \{y,z\}, PT)$
06			
07	PT'	$=$	$ANY(\{z\}, ACCEPT(\{1\}))$
08			$\Box$
09			$NOT(\Sigma_{SYSTEM}, \{z\}, PT')$

Figura 5.5 Cenários distintos no mesmo propósito

- Os contra-exemplos CE_n retornados pela expressão

  $$SYSTEM \sqsubseteq_\tau SYSTEM |[\, \Sigma_{SYSTEM} \,]| PT$$

  são da forma $t \frown \langle mark \rangle$, e $t \in traces(SYSTEM)$, e $mark \in \Sigma_{mark}$ onde $\Sigma_{mark} = \{accept.1, refuse.1\}$, e $SYSTEM \sqsubseteq_\tau t$ (t **conforms** $SYSTEM$). $CE_1 = \langle b, z, refuse.1 \rangle$ e $CE_2 = \langle a, y, b, z, accept.1 \rangle$) são dois exemplos de contra-exemplos.

É possível compor variados cenários combinando os processos primitivos *ACCEPT*, *REFUSE*, *ANY* e *NOT*. Certos cenários por serem demasiadamente comuns, por exemplo, selecionar os cenários onde eventualmente ocorre um evento *ev*, podem ser redefinidos a partir de processos de maior expressividade. *MATCH*, *EXCEPT* e *UNTIL* são processos que possuem uma expressividade maior para definir PTCSPs; que são detalhados nas subseções seguintes.

5.4.3 Processos *MATCH*, *EXCEPT* e *UNTIL*

Uma das construções mais comuns na tarefa de selecionar testes via propósitos de teste é filtrar os cenários onde eventual é encontrado uma evento (ou seqüência) de uma especificação. Como visto anteriormente, através dos processos *ANY* é *NOT* isto é perfeitamente viável, porém não é produtivo. Na Figura 5.5 o processo PT' (linhas 7 a 9) é um exemplo de especificação para o comportamento de $UNTIL(\Sigma_{SYSTEM}, z, ACCEPT(\{1\})$: espera por eventualmente z e se comporta como $ACCEPT(\{1\}$. No exemplo em questão, PT' se comporta aceitando quaisquer outros eventos enquanto z não é oferecido por *SYSTEM* (linha 9), até que eventualmente z é oferecido e sincronizado (linha 7). Uma forma mais concisa e resumida de especificar este comportamento é:

$$PT' = RUN(\Sigma_{SYSTEM} - \{z\}) \triangle ANY(z, ACCEPT(\{1\}))$$

Todos os eventos do alfabeto da especificação com exclusão de z ($\Sigma_{SYSTEM} - \{z\}$) são oferecidos e sincronizados com os eventos comunicados por *SYSTEM* até que esta sincronização seja

interrompida pela ocorrência de um evento do conjunto $\{z\}$ ($\triangle$ $ANY(z, ACCEPT(\{1\}))$) e se comporta como $ACCEPT(\{1\})$. Uma forma mais geral e resumida de especificar $UNTIL(\Sigma, evs, next)$ é apresentada na linha 18 da Figura 5.3.

Pelo emprego de *UNTIL*, fica mais simples especificar cada cenário do PTCSP da Figura 5.5. Na Figura 5.6 encontram-se dois propósitos, cada um especifica um cenário distinto.

$$\begin{aligned} PT2 &= UNTIL(\Sigma_{SYSTEM}, \{y\}, UNTIL(\Sigma_{SYSTEM}, \{z\}, ACCEPT(1))) \\ PT3 &= UNTIL(\Sigma_{SYSTEM}, \{z\}, REFUSE(1)) \end{aligned}$$

Figura 5.6 Exemplo com UNTIL

Em $PT2$, quando se emprega o aninhamento dos processos

$$UNTIL(..., UNTIL(..., ACCEPT(1)))$$

significa que só são marcados os cenários relacionados ao mesmo tempo ao primeiro e ao segundo *UNTIL*: só são considerados os traces onde eventualmente ocorre *y*, e em seguida eventualmente ocorre *z*, não são considerados traces onde ocorra um ou outro separadamente ou, onde ocorram em ordem diversa. $PT2$ retorna o contra-exemplo $CE_1 = \langle a, y, b, z, accept.1 \rangle$. Em $PT3$, são selecionados os cenários onde eventualmente ocorre *z*, e retorna $CE_2 = \langle b, z, refuse.1 \rangle$. Ambos os processos $PT2$ e $PT3$ possuem as propriedades requeridas para uma PTCSP.

Os cenário de $TP2$ utiliza apenas dois conjuntos de eventos ($\{y\}$ e $\{z\}$) para especificar, entretanto *n* conjuntos eventos podem ser útilizados, aumentando indefinidamente a quantidade de processos *UNTIL* aninhados.

Seguindo o raciocínio pode-se pensar em combinar $PT2$ e $PT3$ através de uma escolha externa para obter um PTCSP equivalente a 5.5. Infelizmente, isto não é aconteçe.

$$\begin{aligned} PT4 &= UNTIL(\Sigma_{SYSTEM}, \{y\}, UNTIL(\Sigma_{SYSTEM}, \{z\}, ACCEPT(1))) \\ &\quad \Box \\ &\quad UNTIL(\Sigma_{SYSTEM}, \{z\}, REFUSE(1)) \end{aligned}$$

A escolha externa entre $TP2$ e $TP3$ forma um novo processo não determinístico $PT4$ que produz cenários de teste inconsistente de acordo com as especificações dos cenários (1) e (2) da Seção 5.4.2, por exemplo, $CE_2 = \langle a, y, b, z, refuse.1 \rangle$ é obtido da especificação *SYSTEM*. CE_2 é inconsistente o cenário $\langle a, y, b, z \rangle$ é de aceitação (e deveria ser marcado com $accept.1$) de acordo com (1), entretando o contra-exemplo aponta que o mesmo é um cenário de refutação (está marcado com $refuse.1$). O não determinismo dessa combinação fica explícito pela reescrita do seu comportamento em termos dos processos *RUN* e *ANY* (de acordo com a definição de *UNTIL* da Figura 5.3 linha 18).

$$\begin{aligned} PT4 &= RUN(\{a, b, c, x, y, z\} - \{y\}) \triangle ANY(\{y\}, ...) \\ &\quad \Box \\ &\quad RUN(\{a, b, c, x, y, z\} - \{z\}) \triangle ANY(\{z\}, REFUSE(\{1\})) \end{aligned}$$

Resolvendo a diferença entre conjuntos chega-se:

$$\begin{aligned} PT4 \;=\; & RUN(\{a,b,c,x,z\}) \triangle ANY(\{y\},...) \\ & \Box \\ & RUN(\{a,b,c,x,y\}) \triangle ANY(\{z\}, REFUSE(\{1\})) \end{aligned}$$

O processo os dois processos *RUN* combinados com escolha externa, oferecem simultâneamente cada um dos evento do conjunto $\{a,b,c,x\}$, o que provoca uma comportamento não determinístico. De uma forma geral, a escolha externa entre processos *UNTIL* causa não determinismo. A saída é procurar uma combinação de *ANY* e *NOT* que realize a seleção desejada sem abrir mão do comportamento determinístico. O PTCSP da Figura 5.5 atende aos requisitos para PTCSP (inclusive ser determinístico) e seleciona ambos os cenários.

Quando se pretende selecionar cenários onde o próximo evento oferecido coincida com um evento *ev* tal que $ev \in evset$, emprega-se o processo $MATCH(\Sigma, evset, next)$ como valor para o parâmetro de continuação (*next*) de algum outro processo da Figura 5.3. A linha 14 (da mesma Figura) contém a especificação para o comportamento deste processo que tenta sincronizar com o próximo evento do conjunto *evset* oferecido pela especificação, e se comporta como o processo *next* (o que corresponde ao comportamento de $ANY(evset, next)$), caso contrário, sincroniza como outros eventos oferecidos pela especificação e volta a se comportar como o propósito de teste *initial* (o que corresponde ao comportamento de $NOT(alfa, evs, initial)$).

O exemplo de propósito abaixo mostra como *MATCH* captura cenários onde se espera eventualmete *y*, e imediatamente depois *c*.

$$PT5 = UNTIL(\Sigma_{SYSTEM}, \{y\}, MATCH(\Sigma_{SYSTEM}, \{c\}, ACCEPT(\{2\}), PT5))$$

$MATCH(\Sigma_{SYSTEM}, \{c\}, ACCEPT(\{2\}), PT5)$ foi empregado como valor para o parâmetro *next* do processo $UNTIL(\Sigma_{SYSTEM}, \{y\}, ...)$, por conseguite sincroniza apenas nos cenários nos quais após um eventual *y* o próximo evento oferecido é *c*. $CE_1 = \{c, y, c, accept.2\}$ é o contra-exemplo obtido da seleção de $PT5$ na especificação *SYSTEM*.

De forma adversa, quando é utilizado como continuação de um outro processo, *MATCH* tem comportamento idêntico a *UNTIL* quando utilizado como prefixo de um propósito de teste. Isto pode ser verificado no exemplo abaixo:

Seja

$$PT6 \;=\; MATCH(\Sigma_{SYSTEM}, z, ACCEPT(2), PT6)$$

e

$$PT7 \;=\; UNTIL'(\Sigma_{SYSTEM}, z, ACCEPT(2))$$

então

$$PT6 \sqsubseteq_{\mathcal{FD}} PT7 \quad \wedge \quad PT7 \sqsubseteq_{\mathcal{FD}} PT6$$

Neste contexto, *MATCH* espera que o próximo evento oferecido pela especificação seja exatamente *z*, se isto não ocorre, se comporta novamente como *MATCH* que continua na espera indefinida pelo mesmo evento até que o mesmo aconteça. Este comportamento é equivalente ao eventual acontecimento do evento que é especificado por *UNTIL*.

O processo *EXCEPT* possui comportamento inverso de *MATCH*. É uma construção útil quando se pretende selecionar cenários onde o próximo evento oferecido seja diferente do evento *ev* tal que $ev \in evset$, neste contexto, é empregado como valor para o parâmetro de continuação (*next*) de algum processo da Figura 5.3. É escrito em termo de *MATCH*, o que pode ser

visto na linha 16 da mesma Figura. Quando sincroniza em algum evento do conjunto *evset* se comporta como o processo inicial (*initial*), no caso contrário, se comporta como o próxima processo passado como parâmetro (*next*). Tem comportamento idêntico a $UNTIL(\Sigma, \Sigma - evs, next)$ quando utilizado como prefixo de um propósito de teste.

O exemplo de propósito abaixo mostra como *EXCEPT* captura cenários onde se espera eventualmete *y*, e imediatamente qualquer evento diferente de *c*.

$$PT8 = UNTIL(\Sigma_{SYSTEM}, \{y\}, EXCEPT(\Sigma_{SYSTEM}, \{c\}, ACCEPT(\{3\}), PT8))$$

$EXCEPT(\Sigma_{SYSTEM}, \{c\}, ACCEPT(\{3\}), PT8)$ foi empregado como valor para o parâmetro *next* do processo $UNTIL(\Sigma_{SYSTEM}, \{y\}, ...)$, por conseguite sincroniza apenas nos cenários nos quais após um eventual *y* o próximo evento oferecido é diferente de *c*. $CE_1 = \{a, y, a, accept.3\}$ é o contra-exemplo obtido da seleção de $PT8$ na especificação *SYSTEM*.

A composição entre os processos *EXCEPT*, *MATCH* e *UNTIL* via escolha externa cria um PTCSP de comportamento não determinístico. Para evitar este comportamento caótico é necessário redefinir os três processos a partir dos processos *ANY* e *NOT* de forma habilidosa com intuito de abolir não determinismo.

5.4.4 Processo *MATCHS*

Utilizando como base os processos *ANY* e *NOT* obtem-se *MATCHS*, uma versão do processo *MATCH* que seleciona cenários baseado em uma seqüência *seq* fornecida. Na Figura 5.4.4 está a especificação para este processo.

01	$MATCHS(\Sigma_P, \langle\rangle, next, initial)$	=	$next$
02	$MATCHS(\Sigma_P, seq, next, initial)$	=	$ANY(\{\text{head}(seq)\}, MATCHS(\Sigma_P, \text{tail}(seq), next, initial))$
04			$\Box$
05			$NOT(\Sigma_P, \{\text{head}(seq)\}, initial)$

Figura 5.7 Especificação de *MATCHS*

Quando utilizado com prefixo de um PTCSP, $MATCHS(\Sigma_P, seq, next, initial)$ sincroniza com cenários de teste da especificação *P*, onde eventualmente ocorre a seqüência de eventos *seq* fornecida. No caso base, onde a sequencia recebida é vazia, o processo se comporta como *next* (linha 1). No caso em que $\#seq > 0$, se a especificação oferece o evento head(*seq*), então sincroniza e se comporta como $MATCHS(\Sigma_P, \text{tail}(seq), next, initial)$ (linha 2), caso contrário, se o evento oferecido pertence ao conjunto $\Sigma_p - \{\text{head}(seq)\}$, então se comporta como *initial* (linha 5). Quando é fornecido como valor para o parâmetro *next* de outro processo, e a especificação imediatamente oferece uma seqüência de eventos idêntica a de *seq*, se comporta como *next*, caso contrário, se algum evento oferecido consecutivamente pela especificação não corresponde a um evento de *seq*, se comporta como *initial*. Dependendo do valor dos parâmetros utilizados em *MATCHS* se comporta de forma semelhante a outros dois processos dantes descritos.

1 $MATCHS(\Sigma_P, \langle ev \rangle, next, initial) \equiv MATCH(\Sigma - P, \{ev\}, next, initial)$

2 $INIT = MATCHS(\Sigma_P, seq, match \rightarrow Stop, INIT) \equiv MATCHRW(seq)$

Na equação 1 acima, quando é fornecido como parâmetro uma seqüência unitária $\langle ev \rangle$, *MATCHS* é equivalente ao processo *MATCH* da Figura 5.3, quando o cojunto *evset* é um cojunto unitário $\{ev\}$. Já na equação 2, quando *INIT* é *MATCHS*(...) com *next* igual a $match \rightarrow Stop$ e *initial* igual a *INIT*, *MATCHS* equivale ao processo *MATCHRW*(*seq*) da Seção 5.2.

Adiante está um PTCSP usado para exemplificar o uso de *MATCHS*.

$$PT9 = UNTIL(\Sigma_{SYSTEM}, \{y\}, MATCHSEQ(\Sigma_{SYSTEM}, \langle a, y \rangle, ACCEPT(\{4\}), PT9))$$

O processo *PT*9 seleciona cenários de *SYSTEM* onde eventualmente ocorre *y*, e imediatamente em seguida ocorre a seqüência $\langle a, y \rangle$. O contra-exemplo CE_1 obtido é igual a

$$\langle a, y, a, y, accept.4 \rangle$$

.

No Apêndice B encontra-se um breve resumo sobre o funcionamento de cada um dos processos apresentados nessa seção.

5.5 Relação cspioco

Nesta seção é definida **cspioco**, uma relação de implementação entre uma especificação modelada por um processo CSP (*P*) e uma implementação. Têm como hipótese que a implementação a ser testada possa ser modelada através de um processo CSP (*IUT*), cujo alfabeto de eventos está particionado em entradas e saídas. O alfabeto da implementação Σ_{IUT} é formado pela união do conjunto dos eventos de entrada Σ_{Ii}, com o conjunto dos eventos de saída Σ_{Io}: $\Sigma_{IUT} = \Sigma_{Ii} \cup \Sigma_{Io}$. De forma análoga, o alfabeto da especificação é o conjunto Σ_P obtido pela união das partições de eventos de entrada e saída: $\Sigma_P = \Sigma_{Pi} \cup \Sigma_{Po}$.

De forma intuitiva, uma implementação *IUT* está **cspioco** conforme com uma especificação *P* (*IUT* **cspioco** *P*), se a implementação possui uma quantidade menor ou igual de saídas que podem ser encontradas na especificação, considerando os traces comuns a ambas.

Antes de entrar na definição de **cspioco** que é dada por meio de um refinamento entre processos, apresenta-se a mesma sob a ótica da semântica operacional de CSP, que é a forma mais comum de se definir relações de implementação na literatura (ver Seção 4.4.1). Como esta relação trabalha considerando entradas e saídas como conjuntos distintos, IOLTSs são empregados para representar o comportamento da especificação e implementação pela definição abaixo.

Definição 5.5.1. *Sejam $p, i \in IOLTS(L)$ que correspondem respectivamente aos processos P (especificação) e I (implementação) então,*

$$i \ \textbf{\textit{cspioco}} \ p =_{def} \ \forall t \in traces(p), \ Out(i \ after \ t) \subseteq Out(p \ after \ t)$$

Na definição 5.5.1, destaca-se que as observações da conformidade são todas baseadas a partir dos traces da especificação (*traces*(*p*)). São considerados apenas os eventos visíveis e excluídas observações de comportamentos como *deadlock*, *livelock* e ausência de saídas (*outputlock*). Como conseqüência, a implementação pode conter mais ou menos eventos de quiescence de forma que sua relação com a implementação não é impactada. Considerando um trace *t* da especificação ($t \in traces(p)$), o conjunto de transições com eventos de saídas encontradas nos estados alcançados no IOLTS de uma implementação (*i*) depois de *t*, deve ser um subconjunto das transições de saída existentes nos estados alcançados no IOLTS da especificação (*p*) depois do mesmo trace, isto é: $Out(i \ after \ t) \subseteq Out(i \ after \ t)$. Caso um trace *t* comunicado pela implementação não exista na especificação ($t \notin traces(p)$), qualquer comportamento em termos de transições com eventos de saídas é aceito para o IOLTS da implementação.

Utilizando processos e refinamentos de CSP, uma implementação representada pelo processo *IUT* está **cspioco** conforme uma especificação representada pelo processo *P* (*IUT* **cspioco** *P*) se o seguinte refinamento é válido:

$$P \sqsubseteq_{\tau} (P \ ||| \ RUN(\Sigma_{Io})) \ |[\ | \ \Sigma_{IUT} \ | \ ]| \ IUT \tag{5.9}$$

O refinamento (5.9) executa as mesmas verificações que 5.5.1, com a distinção que o refinamento funciona no domínio denotacional de CSP, enquanto a 5.5.1 raciocina na semântica operacional de CSP (estados e transições de LTS). Do lado esquerdo de (5.9) está o processo da especificação *P*, e do lado direito um processo que será chamado de *SEI* (saídas extras da implementação)

$$SEI = (P \ ||| \ RUN(\Sigma_{Io})) \ |[\ | \ \Sigma_{IUT} \ | \ ]| \ IUT$$

Caso existam traces produzidos pela implementação, comuns à especificação, cuja continuação produz saídas que não ocorrem na especificação após os mesmos traces, estes serão comunicados por *SEI*. Neste caso, o refinamento não é válido e a implementação não é conforme à especificação, por conseguinte $\neg$(*IUT* **cspioco** *P*). No situação contrária, o refinamento é válido: quando todos os traces produzidos pela implementação, comuns a especificação, produzem saídas que estão contidas na especificação; então *IUT* **cspioco** *P*.

SEI é formado pelo paralelismo síncrono do entrelaçamento (interleaving) da especificação *P* com $RUN(\Sigma_{Io})$ (lado esquerdo do paralelismo) com a implementação *IUT* (lado direito do paralelismo). A especificação oferece traces $t \in trace(P)$ que sincronizam evento a evento com a implementação, quando $t \in traces(IUT)$. Se a continuação do comportamento de *t* na implementação origina um trace t' igual a $t \frown \langle evout \rangle$, onde $evout \in \Sigma_{Io}$ e $t \frown \langle evout \rangle \notin traces(P)$, o sufixo $\langle evout \rangle$ ocorre no comportamento da implementação após *t* sincronizado com um evento oferecido pelo processo $RUN(\Sigma_{Io})$. A função de $RUN(\Sigma_{Io})$ é oferecer eventos de saída do alfabeto da implementação para que os sufixo $\langle evout \rangle$ dos traces t', que não sincronizam com *P*, pois $t' \notin trace(P)$, possam sincronizar com *IUT* e ser produzidos por *SEI*. Quando a implementação está conforme à especificação, *SEI* não produz traces $t' = t \frown \langle evout \rangle$, então o refinamento (5.9) é válido e *IUT* **cspioco** *P*. No caso oposto, quando a implementação não está conforme à especificação, *SEI* produz traces $t' = t \frown \langle evout \rangle$, então o refinamento (5.9) é inválido e $\neg$(*IUT* **cspioco** *P*).

Na Figura 5.8 está o processo $IUT1$ (linha 16) que é uma implementação candidata para a especificação *SYSTEM* da Seção 5.1. O processo $IUT1$ se diferencia de *SYSTEM* pela exclusão dos eventos internos (t) e remoção dos trechos $t \rightarrow S0$ em $S2$ e $t \rightarrow S4$ em $S4$. Outras modificações são a inclusão do processo $S10$ (linha 14) e a modificação do processo $S2$ (linha 08). O processo $S10$ comunica os eventos d e y e em seguida se comporta como $S4$. A modificação em $S2$ consiste na adição de uma escolha externa com o processo $S10$.

01	$channel\, d$
02	
03	$AlfaIi1 = \{a, b, c, d\}$
04	$AlfaIo1 = \{x, y, z\}$
05	$AlfaI1 = AlfaIi1 \cup AlfaIo1$
06	
07	$S0 = S9 \,\Box\, S2 \,\Box\, S10$
08	$S2 = c \rightarrow S6 \,\Box\, b \rightarrow S4$
09	$S4 = z \rightarrow S2 \,\Box\, S8$
10	$S6 = y \rightarrow S7$
11	$S7 = c \rightarrow S6$
12	$S8 = y \rightarrow S0$
13	$S9 = a \rightarrow S8$
14	$S10 = d \rightarrow y \rightarrow S4$
15	
16	$IUT1 = S0$

Figura 5.8 Especificação da Implementação IUT1

Empregando o refinamento (5.9) no contexto onde a especificação é *SYSTEM* e a implementação é $IUT1$ tem-se:

$$SYSTEM \sqsubseteq_{\tau} (SYSTEM \;|||\; RUN(AlfaIo1)) \;|[\,|\, AlfaI1 \,|\,]|\; IUT1$$

O resultado da verificação do refinamento acima é verdadeiro; portanto, $IUT1$ é uma implementação válida para *SYSTEM*, de acordo com **cspioco**: $IUT1$ **cspioco** *SYSTEM*. Apesar das diferenças de comportamento encontradas na implementação, sendo a mais significativa a adição dos traces $\langle d, y, ... \rangle$, o processo não quebra as regras da relação de implementação. Tais traces não existem na especificação e, portanto, não infringem a relação de implementação.

Já na Figura 5.9, está o processo $IUT2$ (linha 15) que é uma implementação candidata para a especificação *SYSTEM*. O processo $IUT2$ se diferencia de *SYSTEM* pela exclusão dos eventos internos (t) e remoção dos trechos $t \rightarrow S0$ em $S2$ e $t \rightarrow S4$ em $S4$ (linhas 7 e 8). Outra diferença é a modificação do processo $S8$ (linha 12). Este último processo é modificado pela adição de uma escolha externa com o prefixo $x \rightarrow S0$.

Empregando o refinamento (5.9) no contexto onde a especificação é *SYSTEM* e a implementação é $IUT2$, tem-se:

01	$channel\, d$
02	
03	$AlfaIi2 = \{a, b, c, d\}$
04	$AlfaIo2 = \{x, y, z\}$
05	$AlfaI2 = AlfaIi2 \cup AlfaIo2$
06	
07	$S0 = S9 \,\Box\, S2$
08	$S2 = c \rightarrow S6 \,\Box\, b \rightarrow S4$
09	$S4 = z \rightarrow S2 \,\Box\, S8$
10	$S6 = y \rightarrow S7$
11	$S7 = c \rightarrow S6$
12	$S8 = y \rightarrow S0 \,\Box\, x \rightarrow S0$
13	$S9 = a \rightarrow S8$
14	
15	$IUT2 = S0$

Figura 5.9 Especificação da Implementação IUT2

$$SYSTEM \sqsubseteq_{\tau} (SYSTEM \;|||\; RUN(AlfaIo2)) \;|[\,|\; AlfaI2 \;|\,]|\; IUT2$$

O resultado da verificação do refinamento acima é falso; portanto, $IUT2$ não é uma implementação válida para *SYSTEM*, de acordo com **cspioco**: $\neg(IUT2$ **cspioco** $SYSTEM)$. A relação de implementação não é obedecida pois a implementação comunica o evento *a* de entrada e em seguida o eventos *x* de saída; entretanto, a especificação comunica *a* e como saída não pode comunicar *x* (só pode *y*). Esta violação ocorre porque para o trace $\langle a \rangle$ comum entre implementação e especificação, o evento de saída da implementação *x*, oferecido em seguida, não pode ser oferecido pela especificação após o mesmo evento. Com conseqüência, $IUT2$ não é uma implementação válida.

5.6 Construção de casos de teste em CSP baseados em cenários

A Seção 5.4 descreve uma série de processos utilizados para obter cenários de teste guiados via propósitos de teste CSP. Esta seção mostra como obter casos de teste CSP a partir dos cenários de teste. A abordagem de construção de casos de teste CSP produz testes coerentes (ver Seção 4.4) de acordo com a relação de implementação **cspioco** da Seção 5.5.

5.6.1 Execução, vereditos e coerência

Um caso de teste CSP (CTCSP) é um processo de alfabeto Σ_{CTCSP} que interage com uma implementação, representada por *IUT*, de forma síncrona para aferir se esta última está conforme a especificação *P* (IUT **cspioco** P). O modelo de execução do teste sobre uma implementação (*IUT*) se comporta de acordo com a seguinte expressão:

$$EXEC \;=\; IUT \;\; |[\,|\,\Sigma_{IUT}\,|\,]|\;\; CTCSP \tag{5.10}$$

Em (5.10), o comportamento da implementação é totalmente arbitrário, e CTCSP deve estar preparado para sincronizar com qualquer seqüência de eventos oferecidos pelo *IUT* e assinalar um veredito. Apesar de que no paralelismo a comunicação dos eventos entre processos é síncrona, e não existe direção de origem ou fim (ver paralelismo alfabetizado), conceitualmente, existe uma interpretação de que casos de teste CSP fornecem entradas para o *IUT* cujas saídas são entradas para o CTCSP. O contrário também é verdade; saídas produzidas pela implementação são entradas para o CTCSP, e vice-versa. No sentido direto e usual do teste, o CTCSP fornece eventos (que pertencem ao alfabeto de entrada da implementação) que alimentam a implementação que, em contrapartida, comunica eventos do alfabeto de saída que correspondem às entradas para o CTCSP.

Um CTCSP é sempre construído a partir de um cenário de teste (selecionado por PTCSP), logo estará sempre atrelado a este cenário, pondendo assinalar os seguintes vereditos durante a execução contra uma implementação:

- Passou: a implementação está de acordo com o cenário da especificação selecionado por PTCSP;
- Falhou: a implementação não está de acordo com o cenário da especificação selecionado por PTCSP; e,
- Inconclusivo: a implementação está de acordo com a especificação até certo ponto, porém não é possível aferir se está correta com relação ao cenário de teste selecionado por PTCSP.

Um caso de teste quando executado sobre uma implementação pode levar a diferentes vereditos, pois existem diferentes interações entre implementação e teste. Para um mesmo teste, em dado momento uma implementação pode passar, em outro falhar, ou, ainda, ser inconclusiva. O veredito é determinado quando a implementação sob testes entra em deadlock com o caso de teste, logo após o teste comunicar um dos eventos: *pass*, *fail* ou *inconclusive*. $\Sigma_{verdict} = \{pass, fail, inconclusive\}$ é o conjunto de eventos utilizados para sinalizar os vereditos da execução de testes, onde $\Sigma_{verdict} \cap \Sigma_P = \emptyset$. Os processos abaixo exibem o comportamento do CTCSP na situação em que alcançam um veredito.

$$channel \quad pass, fail, inconclusive$$

$$\begin{aligned} PASS &= pass \rightarrow Stop \\ FAIL &= fail \rightarrow Stop \\ INCO &= inconclusive \rightarrow Stop \end{aligned}$$

CTCSP se comporta como *PASS* quando o teste passa, *FAIL* quando o teste falha e *INCO* quando o teste é inconclusivo. Pela expansão do processo *EXEC* em (5.10), é possível obter todas as possibilidades de interação entre implementação e teste. Utilizando a semântica de *Traces* de *EXEC*, que considera os traces obtidos pela expansão deste processo, é possível

descobrir se eventualmente um CTCSP passa, falha ou resulta em inconclusivo para a implementação *IUT*. Por exemplo, se o evento *pass* $\in$ *traces*(*EXEC*), então alguma possibilidade de interação entre teste e implementação resulta no veredito passou. Esta investigação pode ser formulada através do refinamento abaixo:

$$EXEC \setminus \{\Sigma_P\} \sqsubseteq_\tau PASS$$

Levando em conta que o conjunto de traces do processo *PASS* é $\{\langle\rangle, \langle pass\rangle\}$, o refinamento acima verifica se, ocultando os eventos do alfabeto de P (Σ_P) dos traces de *EXEC*, o conjunto de traces resultante contém $\{\langle\rangle, \langle pass\rangle\}$. Caso o refinamento seja válido, o evento *pass* está presente nos traces de *EXEC*; portanto, eventualmente a execução do teste sobre a implementação resulta no veredito passou.

De forma semelhante, outro refinamento parecido pode verificar se eventualmente o CTCSP pode falhar quando executado contra a implementação *IUT*:

$$EXEC \setminus \{\Sigma_P\} \sqsubseteq_\tau FAIL$$

Se o resultado da verificação do refinamento acima é válido, significa dizer que a implementação em algum momento alcança o veredito falhou.

A propriedade mínima para um caso de teste é que o mesmo seja coerente, isto é, nunca falha em uma implementação que esteja correta de acordo com a especificação. Admitindo que *IUT* **cspioco** *P* e o CTCSP é coerente, eventualmente deve ser possível atingir o veredito passou, e eventualmente não deve ser possível atingir o veredito falha. A propriedade de coerência (ver (4.11)) para um CTCSP pode ser formulada a partir dos dois refinamentos anteriores.

Definição 5.6.1. *Seja uma implementação IUT e uma especificação P, tal que IUT* ***cspioco*** *P, e seja* Σ_P *o alfabeto de eventos da especificação. Um CTCSP construído a partir da especificação P é coerente se os refinamentos a seguir podem ser verificados como verdade:*

$$EXEC \setminus \{\Sigma_P\} \sqsubseteq_\tau PASS$$
$$EXEC \setminus \{\Sigma_P\} \not\sqsubseteq_\tau FAIL$$

A definição (5.6.1) manifesta que as execuções de um processo *CTCSP* contra uma implementação em conformidade, *IUT*, devem eventualmente passar (de acordo com o primeiro refinamento da definição), e nunca falhar (de acordo com o segundo refinamento da definição).

5.6.2 Processo *TC_BUILDER*

A partir de um cenário de teste CE_n obtido por algum propósito de teste (ver Seção 5.4), podemos construir um CTCSP fornecendo os parâmetros adequados para o processo *TC_BUILDER* (linhas 1,2 e 3) da Figura 5.6.2.

O processo *TC_BUILDER*(*ialfa*, *oalfa*, *s*) recebe como parâmetros o alfabeto de entrada da implementação (*ialfa*), o alfabeto de saída da implementação (*oalfa*) e uma seqüência de tuplas (*s*) que é estruturada a partir do cenário de teste CE_n. A seqüência *s* possui o seguinte formato:

$$s = \langle (element(1, CE_n), initials_1), ..., (element(\#CE_n, CE_n), initials_N) \rangle$$

01	$TC\_BUILDER(\Sigma_{Ii}, \Sigma_{Io}, \langle(accept.i,)\rangle) = PASS$
02	$TC\_BUILDER(\Sigma_{Ii}, \Sigma_{Io}, s) = SUBTC(\Sigma_{Ii}, \Sigma_{Io}, \text{head}(s))$;
03	$TC\_BUILDER(\Sigma_{Ii}, \Sigma_{Io}, \text{tail}(s))$
04	$SUBTC(\Sigma_{Ii}, \Sigma_{Io}, (ev, initial)) =$
05	$if(ev \in \Sigma_{Ii})$ *then*
06	$ev \rightarrow SKIP$
07	*else*
08	$ev \rightarrow SKIP$
09	$\square$
10	$ANY(initial - \{ev\}), INCO)$
11	$\square$
12	$NOT(\Sigma_{Ii} \cup \Sigma_{Io}, initial \cup \{ev\}, FAIL)$

Figura 5.10 Processo construtor para casos de teste CSP

Cada tupla $((element(N, CE_n), initials_N)$ de s, onde $1 \leq N \leq \#CE_n$, possui o primeiro elemento igual ao n-ésimo elemento encontrado no cenário de teste ($element(N, CE_n)$), e o segundo elemento ($initials_N$) corresponde ao conjunto dos outros eventos oferecidos pela especificação no mesmo ponto da especificação onde é oferecido o n-ésimo elemento de CE_n.

Pode-se criar uma instância de s tomando como exemplo o cenário de teste $CE_2 = \langle a, y, b, z, accept.1\rangle$ (pois CE_1 é um cenário excluído da seleção) selecionado da especificação *SYSTEM* pela propósito de teste *PT* da Figura 5.5. Considerando a especificação *SYSTEM* e o cenário CE_2, tem-se s_{CE1}, que funciona como a seqüência s no caso particular de especificação e cenário de teste deste exemplo:

$$s_{CE1} = \langle(a, \emptyset), (y, \emptyset), (b, \emptyset), (z, \emptyset), (accept.1, \emptyset)\rangle$$

O valor do parâmetro $initials_N$ para todas as tuplas é inicialmente vazio, sendo preenchido em uma etapa posterior do processo de construção do CTCSP. A função deste parâmetro pode ser entendida a partir do comportamento dos processos *TC_BUILDER* e *SUBTC*. O primeiro processo é definido como a composição seqüencial do segundo, que trata cada tupla de s é tratada individualmente.

TC_BUILDER se comporta como o processo *PASS* quando s equivale a $\langle(accept.i,)\rangle$ (a última tupla a ser encontrada), neste caso, o teste alcança o veredito passou. Porém, antes de assinalar este veredito, o teste precisa analisar as tupla anteriores de s, na ordem em que são encontradas, para validar o funcionamento da implementação com respeito a ocorrência do cenário de teste CE_n. *TC_BUILDER* se comporta como a composição seqüencial especificada nas linhas 2 e 3 da Figura 5.6.2. Tal composição se comporta primeiramente como o processo *SUBTC*, que recebe como parâmetro a tupla corrente de s (head(s) na linha 2). No caso de terminação com sucesso, *SUBTC* continua como o processo *TC_BUILDER*, quando este último recebe como parâmetro uma nova seqüência, cujos elementos correspondem as tuplas posteriores da mesma seqüência (tail(s) na linha 3).

O processo *SUBTC* (linhas 4 a 12) funciona como um teste fundamental (ou sub teste), que trabalha centrado em cada tupla fornecida pelo processo *TC_BUILDER*. O papel de *SUBTC* é confrontar os eventos oferecidos pela implementação *IUT* com os valores especificados no CTCSP. Para tanto, os valores da tupla $(ev, initial)$ são considerados para produzir um veredito apropriado. Considerando os alfabetos de entrada e saída da implementação (Σ_{Ii} e Σ_{Io}), este processo assume comportamentos distintos dependendo se o evento *ev* pertence a um ou ao outro alfabeto:

- Se $ev \in \Sigma_{Ii}$: Neste ponto da execução, quando *ev* pertence ao alfabeto de entrada da implementação, isto é, consiste num evento fornecido pelo teste como entrada para a implementação, nenhuma resposta da implementação é esperada. Logo, *SUBTC* oferece *ev* como evento e termina com sucesso;

- Se $ev \in \Sigma_{Io}$: Sendo *ev* um evento específico esperado como saída da implementação, outros eventos diferentes de *ev* podem ser comunicados pela implementação. No caso em que a resposta de implementação coincide com *ev*, *SUBTC* sincroniza neste evento e termina com sucesso (linha 8). Para outros casos, quando o evento comunicado *iout* é diferente de *ev*, existem duas situações a lidar:

  - $iout \in initial$: O evento *iout* pertence ao conjunto *initial*. Neste contexto, apesar do evento oferecido (*iout*) ser diferente do evento esperado (*ev*) para o cenário que originou o teste, a implementação comunica uma ação prevista na especificação, conseqüentemente é assinalado o veredito inconclusivo (linha 10); e,
  - $iout \notin initial$: O evento *iout* não pertence ao conjunto de aceitação *initial*. Neste caso, *iout* é um elemento do alfabeto da implementação Σ_P que de acordo com especificação, não está autorizado a ocorrer neste ponto do cenário de teste. Para estes tipos de eventos é assinalado o veredito falha (linha 12).

Usando como exemplo a seqüência s_{CE1}, é possível construir o CTCSP abaixo:

$$CT = TC\_BUILDER(\Sigma_{SYSTEMi}, \Sigma_{SYSTEMo}, s_{CE1})$$

O processo *CT* corresponde a um caso de teste concebido do cenário de teste CE_1 obtido da especificação *SYSTEM*. Utilizando como implementação trivial a própria especificação (*IUT* = *SYSTEM*), de acordo com a definição (5.6.1), pode-se verificar que este caso de teste não é coerente.

Seja $EXEC_{SYSTEM} = SYSTEM \ |[\,|\,\Sigma_{SYSTEM}\,|\,]|\ CT$ então:
$EXEC_{SYSTEM} \setminus \{\Sigma_{SYSTEM}\} \sqsubseteq_\tau PASS$
$EXEC_{SYSTEM} \setminus \{\Sigma_{SYSTEM}\} \sqsubseteq_\tau FAIL$

O segundo refinamento acima, mostra que *CT* eventualmente pode redundar em falha se executado contra a própria especificação, quando isto não deveria acontecer. Por conseguinte, *CT* não é coerente.

A expansão do processo *CT* possui o mesmo comportamento do processo CT_0 especificado a seguir.

$$\begin{aligned} CT_0 &= a \rightarrow CT_1 \\ CT_1 &= y \rightarrow CT_2 \,\Box\, ANY(\Sigma_{SYSTEMo} - \{y\}, FAIL) \\ CT_2 &= b \rightarrow CT_3 \\ CT_3 &= z \rightarrow PASS \,\Box\, ANY(\Sigma_{SYSTEMo} - \{z\}, FAIL) \end{aligned}$$

Analisando a continuação do processo CT_0 (que é a expansão de *CT*), encontramos o comportamento que ocasiona uma falha quando o teste é executado contra a especificação *SYSTEM*. O processo CT_0 ao chegar a se comportar como CT_3 executa o trace $\langle a, y, b\rangle$ em sincronia com a especificação *SYSTEM*. A opções que este teste espera da especificação como CT_3 são: o evento de saída *z*, que leva o teste a passar, e, qualquer evento de saída que pertence ao conjunto $\Sigma_{SYSTEMo} - \{z\}$, que leva o teste a falhar. Observando a semântica operacional do processo *SYSTEM* na Figura 5.1, fica claro que o comportamento da especificação após o trace $\langle a, y, b\rangle$ aceita a ocorrência do evento de saída *y*, portanto, um teste não pode falhar caso *y* seja produzido por *SYSTEM* neste ponto. Acontece que CT_3 falha de forma indevida quando *y* é produzido, quando deveria resultar no veredito inconclusivo. Portanto *CT* não é coerente.

Em verdade, nem todo CTCSP formado por *TC_BUILDER* é coerente, é preciso ajustar o valor $initials_N$ encontrado nas tuplas do parâmetro *s*. Na próxima subseção é mostrado como realizar o processo este ajuste e tornar o CTCSP coerente.

5.6.3 Tornando o CTCSP coerente

A Seção 5.6.2 explicita a utilidade do parâmetro *s* em *TC_BUILDER*. Dependendo do seu conteúdo, o caso de teste sinaliza o veredito falha ou inconclusivo (linhas 10 a 12 da Figura 5.6.2). Como dito, de início, o valor de $initials_N$ é inicializado como conjunto vazio para todas a tuplas de *s*. Se definido indevidamente, o caso de teste resultante, mesmo para implementações corretas, pode eventualmente resultar em falha (quando deveria ser inconclusivo), o que vai de encontro a propriedade coerente, que impede que haja eventuais falhas em implementações corretas. Esta subseção mostra como atualizar o valor de *s*[1] para tornar coerente os testes obtidos da seção anterior.

Se a execução de um CTCSP eventualmente falha quando executado contra uma implementação trivial (a própria especificação), é porque algum conjunto de aceitação $initials_N$ da seqüência *s* está definido errado. Isto pode ser visualizado analisando detalhadamente o comportamento do processo *TC_BUILDER* que inicialmente se comporta como o processo *SUBTC*. Neste último processo, quando o evento oferecido pela implementação, denotado *iout*, é diferente do evento *ev* (opção da linha 12 na Figura 5.6.2), e não está contido no conjunto *initial* (linhas 8 a 10 da mesma figura), o veredito assinalado pelo CTCSP é falha. Como a implementação, no caso é a própria especificação, o veredito deveria ser inconclusivo, pois o evento *iout* está previsto na especificação para este ponto. No momento em que *iout* é incluído no conjunto $initials_N$, a ocorrência deste evento passa a ser considerada um comportamento esperado.

[1] O esperado seria construir *s* correto por construção, todavia o estágio atual da abordagem requer uma inicialização padrão de *s* com posterior atualização. Fica como um trabalho futuro aperfeiçoar a abordagem neste ponto.

Concluída esta alteração, a execução do CTCSP sobre esta mesma implementação resulta no veredito inconclusivo. Para que o CTCSP seja coerente, todos os eventos *iout* que fazem parte da especificação devem ser incluídos nos conjuntos de aceitação $initials_N$ de s. Para incluir os eventos é necessário primeiramente identificá-los.

Lembrando que $EXEC_{CTCSP} = P[|\ (\Sigma_P\ |]CTCSP$, o contra-exemplo retornado pelo refinamento (5.11) indica por onde iniciar a atualização do conjunto $initials_N$.

$$EXEC_{CTCSP} \sqsubseteq_\tau EXEC_{CTCSP}[|\ \Sigma_P \cup \{fail\}\ |]UNTIL(\Sigma_P, \{fail\}, REFUSE(\{1\})) \tag{5.11}$$

No refinamento (5.11), o processo $EXEC_{CTCSP}$ corresponde à execução do caso de teste sobre a implementação trivial (a especificação P). A construção *UNTIL* é utilizada para selecionar os cenários onde eventualmente ocorre uma falha. Portanto, o referido refinamento quando falso, retorna, caso exista, tal cenário de $EXEC_{CTCSP}$. Este cenário indica o evento que deve ser incluído no conjunto $initials_N$. No caso em que o refinamento é falso, surge um contra-exemplo C_{CTCSP} da forma $t^\frown\langle fail, refuse.1\rangle$, onde t é uma seqüência de eventos que pertence a Σ_P. O valor do comprimento da seqüência t ($\#t$) indica a posição da tupla em s que precisa atualizar $initials_{\#t}$ pela inclusão do evento que está na última posição de t. Desta atualização surge s', que corresponde a s após a atualização de $initials_{\#t}$. De s' pode ser obtido um novo CTCSP ($CTCSP'$) que é executado contra P pela expressão $EXEC_{CTCSP'} = P[|\ (\Sigma_P\ |]CTCSP$. $EXEC_{CTCSP'}$ é reaplicado no refinamento (5.11) substituindo $EXEC_{CTCSP}$ e repetindo as atualizações necessárias para $initials_N$. Esta reaplicação aconteçe até que o refinamento seja válido, portanto nenhum contra-exemplo será retornado. Neste ponto, não há ajustes a ser efetuados no teste, pois o mesmo é coerente.

A Figura 5.11 mostra através de exemplos, o processo completo para obtenção de um caso de teste, bem como a aplicação de (5.11) para identificar se CT é coerente.

01	$PT2 = UNTIL(\Sigma_{SYSTEM}, \{y\}, UNTIL(\Sigma_{SYSTEM}, \{z\}, ACCEPT(1)))$
02	
03	$SYSTEM \sqsubseteq_\tau SYSTEM\,|[\,\Sigma_{SYSTEM}\,]|\,PT2$
04	
05	$CE_1 = \langle a, b, y, z, accept.1\rangle$
06	
07	$s_{CE1} = \langle (a,\emptyset), (y,\emptyset), (b,\emptyset), (z,\emptyset), (accept.1,\emptyset)\rangle$
08	
09	$CT = TC\_BUILDER(\Sigma_{SYSTEMi}, \Sigma_{SYSTEMo}, s_{CE1})$
10	
11	$EXEC_{SYSTEM} = SYSTEM\ \ |[|\ \Sigma_{SYSTEM}\ |]|\ \ CT$
12	
13	$EXEC_{SYSTEM} \sqsubseteq_\tau EXEC_{SYSTEM}[|\ union(\Sigma_{SYSTEM}, fail)\ |]UNTIL(\Sigma_{SYSTEM}, fail, REFUSE(1))$

Figura 5.11 Processo completo para obtenção de um caso de teste e verificação de coerência

Acompanhando a Figura 5.11, na linha 1, tem-se o PTCSP $PT2$ que seleciona cenários de teste onde eventualmente ocorre y e eventualmente ocorre z, pelo evento de marcação $accept.1$. Na linha 3, está um refinamento cujo resultado da verificação é falso quando aplica $PT2$ sobre a especificação *SYSTEM* para obter o cenário de teste CE_1 (da linha 5). A partir deste cenário é formado o parâmetro s_{CE1} (na linha 7) que é utilizado como parâmetro de *TC_BUILDER* para dar comportamento ao CTCSP *CT* (linha 09). Na linha 11, está a execução formal entre especificação e CTCSP. Finalmente, na linha 13, está o refinamento (5.11) que indica como completar s_{CE1}, de forma que uma nova versão do teste *CT* torne-se coerente. Este refinamento retorna o contra-exemplo $C_{CT} = \langle a, y, b, y, fail, refuse.1 \rangle$ que indica que o conjunto $initials_4$ de s_{CE1} precisa incluir o evento y. Abaixo está a seqüência s'_{CE1} que corresponde a s_{CE1} depois do ajuste em $initials_4$.

$$s'_{CE1} = \langle (a,\emptyset),(y,\emptyset),(b,\emptyset),(z,\{y\}),(accept.1,\emptyset) \rangle$$

A partir de s'_{CE1} controe-se uma novo caso de teste, uma nova execução de teste, e reaplica-se (5.11) para verificar se o novo teste é coerente.

$$CT' = TC\_BUILDER(\Sigma_{SYSTEMi}, \Sigma_{SYSTEMi}, s'_{CE1})$$
$$EXEC'_{SYSTEM} = SYSTEM \ |[\,|\,\Sigma_{SYSTEM}\,|\,]|\ CT'$$
$$EXEC_{SYSTEM} \sqsubseteq_{\tau} EXEC'_{SYSTEM}[|\ union(\Sigma_{SYSTEM}, fail)\ |]UNTIL(\Sigma_{SYSTEM}, fail, REFUSE(1))$$

Na especificação acima, CT' representa o novo CTCSP obtido a partir de s'_{CE1}.

A expansão do processo CT' possui o mesmo comportamento do processo CT'_0 especificado a seguir.

$$\begin{aligned}
CT'_0 &= a \rightarrow CT'_1 \\
CT'_1 &= y \rightarrow CT'_2 \,\Box\, ANY(\Sigma_{SYSTEMo} - \{y\}, FAIL) \\
CT'_2 &= b \rightarrow CT'_3 \\
CT'_3 &= z \rightarrow PASS \,\Box\, ANY(\{y\}, INCO) \,\Box\, NOT(\Sigma_{SYSTEM}, \{z,y\}, FAIL)
\end{aligned}$$

Analisando a continuação do processo CT'_0 (que é a expansão de CT'), verificamos que este corresponde a uma modificação do processo CT_0 (da Seção 5.6.2) pelas mudanças no processo CT'_3. Entretanto, CT'_0 é coerente e CT_0 não o é. O processo de teste CT'_0 durante a execução contra a especificação *SYSTEM*, quando não encontra o cenário de teste especificado, porém encontra um comportamento que faz parte da especificação não falha, pois o veredito coerente para este comportamento é inconclusivo. Isto pode ser verificado usando como exemplo o comportamento de CT'_0 que ao chegar a se comportar como CT'_3 executa o trace $\langle a, y, b \rangle$ em sincronia com a especificação *SYSTEM*. A opções que este teste espera da especificação como CT'_3 são: o evento de saída z, que leva o teste a passar, o evento de saída y, que leva o teste a ser inconclusivo, e, qualquer evento do alfabeto diferente de z e y, que leva o teste a falhar. Observando a semântica operacional do processo *SYSTEM* na Figura 5.1, fica claro que o comportamento da especificação após o trace $\langle a, y, b \rangle$ aceita a ocorrência do evento de saída y, portanto, um teste deve ser inconclusivo no caso em que y aconteça.

Continuando a verificar a coerência de CT' através de refinamentos, uma nova execução $EXEC'_{SYSTEM}$ é montada com CT', e por último o refinamento que indica se CT' pode eventualmente falhar. O resultado do refinamento é positivo, o que indica que não existem mais

ajustes a realizar no caso de teste CT', pois o mesmo é um caso coerente - jamais falha para uma implementação correta. A Figura 5.13 mostra o processo completo para obtenção do caso de teste coerente CT'.

01	$PT2 = UNTIL(\Sigma_{SYSTEM}, \{y\}, UNTIL(\Sigma_{SYSTEM}, \{z\}, ACCEPT(1)))$
02	
03	$SYSTEM \sqsubseteq_\tau SYSTEM \,|[\, \Sigma_{SYSTEM} \,]|\, PT2$
04	
05	$CE_1 = \langle a, b, y, z, accept.1 \rangle$
06	
07	$s_{CE1} = \langle (a,\emptyset), (y,\emptyset), (b,\emptyset), (z,\emptyset), (accept.1,\emptyset) \rangle$
08	
09	$CT = TC\_BUILDER(\Sigma_{SYSTEMi}, \Sigma_{SYSTEMi}, s_{CE1})$
10	
11	$EXEC_{SYSTEM} = SYSTEM \;\; |[\,|\, \Sigma_{SYSTEM} \,|\,]|\;\; CT$
12	
13	$EXEC_{SYSTEM} \sqsubseteq_\tau EXEC_{SYSTEM}[|\, union(\Sigma_{SYSTEM}, fail) \,|]UNTIL(\Sigma_{SYSTEM}, fail, REFUSE(1))$
14	
15	$s'_{CE1} = \langle (a,\emptyset), (y,\emptyset), (b,\emptyset), (z,\{y\}), (accept.1,\emptyset) \rangle$
16	
17	$CT' = TC\_BUILDER(\Sigma_{SYSTEMi}, \Sigma_{SYSTEMo}, s'_{CE1})$
18	
19	$EXEC'_{SYSTEM} = SYSTEM \;\; |[\,|\, \Sigma_{SYSTEM} \,|\,]|\;\; CT'$
20	
21	$EXEC_{SYSTEM} \sqsubseteq_\tau EXEC'_{SYSTEM}[|\, union(\Sigma_{SYSTEM}, fail) \,|]UNTIL(\Sigma_{SYSTEM}, fail, REFUSE(1))$

Figura 5.12 Processo completo para obtenção de um caso de teste coerente

Na especificação que sucede, é possível analisar os eventuais vereditos encontrados quando se executa CT' na implementação $IUT1$ da Seção 5.5.

$EXEC\_IUT1 = IUT1 \,|[\,|\, AlfaI1 \,|\,]|\, CT'$
$EXEC\_IUT1 \setminus AlfaI1 \sqsubseteq_\tau pass \rightarrow STOP$
$EXEC\_IUT1 \setminus AlfaI1 \sqsubseteq_\tau fail \rightarrow STOP$

O resultado do primeiro refinamento é verdadeiro, pois eventualmente o caso de teste CT' passa em uma das possíveis interações com $IUT1$. Já o segundo é falso, porque sendo a $IUT1$ **cspioco** $SYSTEM$, e o caso de teste coerente, não existe uma execução na qual o caso de teste falhe.

Em outro exemplo, emprega-a implementação $IUT2$ da Seção 5.5 para analisar os vereditos eventualmente alcançados quando executando o caso de teste CT'.

$EXEC\_IUT2 = IUT2 \,|[\,|\, AlfaI2 \,|\,]|\, CT'$
$EXEC\_IUT2 \setminus AlfaI1 \sqsubseteq_\tau pass \rightarrow STOP$
$EXEC\_IUT2 \setminus AlfaI1 \sqsubseteq_\tau fail \rightarrow STOP$

Apesar de ¬(*IUT*2 **cspioco** *SYSTEM*), *CT′* alcança com *IUT*2 eventualmente o mesmo veredito de quando é executado contra *IUT*1. Isto acontece, pois sendo este teste coerente, é garantido que ele não falha em especificações corretas, entretanto, pode aceitar especificações não conformes - que é o caso de *IUT*2.

CAPÍTULO 6

Estudo de Caso

Neste capítulo apresentamos a aplicação prática dos resultados obtidos no capítulo anterior em um subconjunto das especificações de aparelhos celulares proveniente de uma parceria entre o Centro de Informática e a Motorola Industrial Limitada [SAV+05]. As especificações foram disponibilizadas pelo projeto de testes, fruto dessa parceria, denominado de BTC (Brazilian Test Center).

Os experimentos apresentados foram produzidos por uma ferramenta, denominada de ATG (Abstract Test Generation), desenvolvida no próprio contexto deste trabalho. Através desta ferramenta foi possível gerar automaticamente vários casos de teste seguindo a abordagem apresentada no capítulo anterior.

Os casos de teste gerados automaticamente foram então comparados com os testes gerados manualmente a partir dos mesmos requisitos.

Além de apresentar os casos de teste produzidos pela nossa ferramenta ATG, também conduzimos um trabalho comparativo sobre os casos de teste gerados automaticamente seguindo nossa contribuição e os casos de teste originais do BTC, criados manualmente.

Como visto no capítulo anterior, tanto requisitos quanto casos de teste são descritos formalmente usando a notação CSP. Porém, neste capítulo todos os fragmentos de CSP são apresentados em sua versão de máquina, denominada de CSP_M, que é manipulada pelo verificador de modelos FDR (maiores detalhes sobre CSP_M podem ser encontrados na Seção 3.4).

6.1 Ferramenta ATG

A ferramenta ATG (Abstract Test Generation) é uma ferramenta acadêmica que implementa a abordagem de geração proposta no Capítulo 5. A Figura 6.1 mostra a estrutura de entradas e saídas da ferramenta e sua interação com FDR (maiores detalhes sobre FDR podem ser encontrados na Seção 3.4). Nesta figura as setas estão rotuladas indicando quais atividades são realizadas usando o suporte de FDR e quais são realizadas por ela própria ATG. A elaboração de processos e expressões de refinamento são realizadas unicamente por ela ATG mas, a verificação dos refinamentos elaborados (empregada para análise de consistência de propósitos de teste e geração de testes) é feita por FDR.

Na Figura 6.1 vê-se que ATG recebe como entrada o processo inicial da especificação de comportamento (denotada PC), pseudo propósitos de teste CSP (PPT) e algumas informações de ajuste para geração (AG). Pseudo propósitos de teste são especificações CSP que definem cenários de seleção de testes de forma equivalente aos propósitos de teste CSP reais (Seção 5.4). Como exemplo de pseudo propósitos, na Figura 6.2 temos os processos $PPT2$ e $PPT3$.

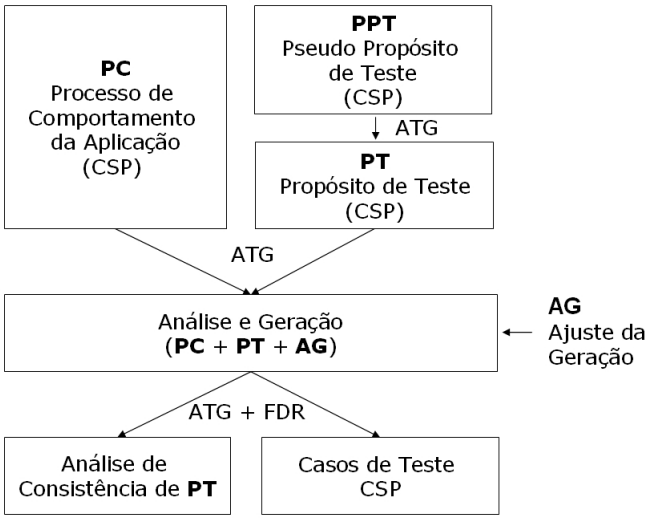

Figura 6.1 ATG - Gerador de Testes Abstratos

Esses processos descrevem de forma abstrata (porém não substituem) os propósitos de teste $PT2$ e $PT3$ da Figura 5.6.

$$PPT2 = un.\{y\} \rightarrow un.\{z\} \rightarrow ac.\{1\} \rightarrow Skip$$
$$PPT3 = un.\{z\} \rightarrow re.\{1\} \rightarrow Skip$$

Figura 6.2 Pseudo CSP para propósito de testes

A ferramenta transforma os pseudo propósitos de teste (PPT) em propósitos de teste reais (PT). O formato dos pseudo propósitos de teste, bem como o processo de transformação para propósitos de teste CSP reais, é mostrado em detalhes na Seção 6.1.1.

ATG montada e envia para FDR expressões de refinamentos CSP (ver Seção 5.3) formadas a partir do processo inicial da especificação (PC), do propósito de teste CSP (PT) e das informações de ajuste (AG). FDR verifica e devolve os resultados da verificação. ATG utiliza os resultados para gerar os casos de teste CSP e fazer análise de consistência dos propósitos de teste. A quantidade de testes a ser gerados é um parâmetro de ajuste (AG) para a geração. A análise de consistência identifica se um propósito, ou um conjunto deles, está consistente com a especificação CSP fornecida, isto é, se são capazes de gerar algum caso de teste. Se o cenário descrito pelo propósito é válido, significa que ao menos um caso de teste poderá ser obtido.

A principal funcionalidade de ATG é gerar casos de testes CSP, onde o processo é guiado por propósitos de teste. Entretanto, existem outras funcionalidades igualmente úteis, tais como:

Verificar as propriedades do processo É possível verificar as propriedades básicas como *deadlock*, *livelock* e determinismo dos processos da especificação. Como ATG possui comunicação

com FDR, ela disponibiliza uma interface transparente e prática para verificação das propriedades básicas (que são realizadas por FDR). Desta forma, não existe necessidade de instanciar a interface ou a versão por linha de comando de FDR. Esta facilidade propicia um ambiente integrado de geração de testes e verificação das especificações utilizadas como entrada;

Verificar conformidade de testes Um teste pode ser confrontado contra a especificação CSP para verificar se ele **conforms** [LdBB+01] a especificação, isto é, se é um refinamento válido (pelo modelo de traces) da mesma. Isso é particularmente útil quando após uma mudança na especificação deseja-se verificar quais dos testes, anteriormente produzidos, continuam coerentes com a especificação e quais não estão. Os testes que estão coerentes continuam a ser utilizados sem restrições. Já os testes não coerentes, dependendo de sua quantidade, podem ser pontualmente atualizados (gerada uma nova versão), ou, gerado novamente todo o conjunto de testes. No caso em que são poucos os teste que não estão coerentes, é viável utilizar a versão mais atual da especificação com propósitos para obter um conjunto de testes substitutos; e,

Gerar LTS de um processo CSP A ferramenta gera um grafo (LTS) de qualquer processo CSP fornecido (especificações de comportamento do sistema como especificações de testes). Esse grafo constitui uma representação visual (semântica operacional) do modelo CSP, que pode facilitar o entendimento sobre o comportamento de certas porções da especificação e/ou dos testes. O grafo é gerado no formato TGF [yWo06a] que pode ser visualizado pela ferramenta de manipulação de grafos chamada yEd [yWo06b]. Como existem outras ferramentas que trabalham a partir de LTS, esse grafo também pode servir de base para obtenção de outros formatos de grafo utilizados como entrada para ferramentas de geração de testes baseadas em LTSs (ver Seção 4.4).

ATG foi implementada com tecnologia Java versão 1.5 multiplataforma [Mic06]. Possui uma interface em linha de comandos e uma interface gráfica que está sendo aprimorada. Pode ser executada tanto localmente (desde que se esteja usando LINUX [Lin06] com uma instalação de FDR) quanto remotamente, quando ativado o modo servidor.

6.1.1 Pseudo Propósitos de Teste

Esta seção demonstra brevemente como a ferramenta ATG transforma a notação de pseudo propósitos de teste CSP em propósitos de teste CSP reais (os últimos são definidos na Seção 5.4). Pseudo propósitos são utilizados como uma notação mais simples para especificar propósitos de teste reais: descrevem sucintamente um propósito de teste real. Apesar de descrever, pseudo propósitos não substituem os propósitos de teste reais; são apenas uma facilidade de notação. Ora, o que se quer é obter propósitos de teste reais a partir da sua descrição (pseudo). Com este objetivo, ATG obtém uma representação simbólica dos pseudo propósitos que é passada como parâmetro ao processo `TP_BUILDER`, cujo comportamento equivale ao propósito de teste real. Por simplicidade, os passos dessa transformação serão demonstrados utilizando especificações CSP_M.

Podemos construir pseudo propósitos de teste utilizando as declarações da seguinte especificação CSP_M.

```
channel st
channel ac,re : {1..MAX_MARKS}
channel an,no,un,mt,ex : AlfaS
```

Pseudo propósitos são processos descritos em termos do evento `st` e dos eventos que são produzidos pelos canais `ac`, `re`, `an`, `no`, `un`, `mt` e `ex`. O evento `st` representa o comportamento inicial do processo de propósito de teste que se quer representar. Já os canais correspondem aos processos da Figura 5.3 de acordo com o mapeamento mostrado na Tabela 6.1.

`ac`	*ACCEPT*
`re`	*REFUSE*
`an`	*ANY*
`no`	*NOT*
`un`	*UNTIL*
`mt`	*MATCH*
`ex`	*EXCEPT*

Tabela 6.1 Mapeamento Pseudo Propósito

Na Tabela 6.1, a coluna da esquerda possui o nome do canal do pseudo propósito e a coluna da direita o nome do processo primitivo correspondente. Da declaração de canais apresentada anteriormente, podemos ver que os canais `ac` e `re` comunicam números naturais entre 1 e `MAX_MARKS`, enquanto os canais `an`, `no`, `un`, `mt` e `ex` comunicam conjuntos s tal que $s \subseteq \mathbb{P}\, AlfaS$.

Como exemplo do uso desta notação, podemos construir os seguintes pseudo propósitos de teste `PTP`, `PTP2` e `PTP3`.

```
PPT =  an.{y} -> un.{z} -> ac.1 -> SKIP
       []
       an.{z} -> re.1 -> SKIP
       []
       no.{y,z} -> st -> SKIP

PPT2 = un.{y} -> un.{z} -> ac.1 -> SKIP

PPT3 = un.{z} -> re.1 -> SKIP
```

Esses três pseudo propósitos representam os processos *TP*, *TP*2 e *TP*3 da Seção 5.4.

A partir desta notação, a ferramenta ATG obtém uma representação simbólica definida de acordo com a estrutura de dados (6.1).

$$tps = \langle (t_0, p_0, tps_0), ..., (t_N, p_N, tps_N) \rangle \tag{6.1}$$

Cada tripla da seqüência *tps* representa a codificação de um processo primitivo colocado em escolha externa com outros. O parâmetro t_i (onde $0 \leq i \leq N$ representa o índice do prefixo de um processo que participa da escolha externa) da tripla corresponde ao identificador do processo primitivo. Este parâmetro pode assumir os valores $ac, re, an, no, un, mt, ex$. O parâmetro p_i da

tripla corresponde ao parâmetro do processo primitivo t_i. O parâmetro tps_i representa o valor do parâmetro *next* do processo primitivo (para mais detalhes sobre o parâmetro *next* consultar a Figura 5.3).

Seguindo a estrutura de (6.1), podemos obter as representações simbólicas dos processos PPT, PPT2 e PPT3.

```
PPTS = <
        (an,{y},<(un,{z},<(ac,{1},<>)>)>),
        (an,{z},<(re,{1},<>)>),
        (no,{y,z},<(st,{},<>)>)
        >

PPT2S = <(un,{y},<(un,{z},<(ac,{1},<>)>)>)>

PPT3S = <(un,{z},<(re,{1},<>)>)>
```

A partir de cada estrutura simbólica, ATG vai produzir o processo de propósito real, cuja pseudo representação é fornecida. Tomando como exemplo a estrutura PPT3S, segue o passo a passo do mapeamento.

1. A tripla na parte mais externa da estrutura <(un,{z},...> é transformada no processo primitivo *UNTIL*(*AlfaS*,{*z*},*next*);

2. O terceiro parâmetro da tripla mais externa, representado pela estrutura simbólica <(re,{1},<>)>, resulta no processo *REFUSE*({1}), que é colocado como valor para o parâmetro *next* do processo mais externo *UNTIL*; e,

3. Finalmente, o propósito de teste real obtido a partir de PPT3S é *UNTIL*(*AlfaS*,{*z*},*REFUSE*({1})).

O processo TP_BUILDER tem a função de realizar o mapeamento dos elementos de uma estrutura simbólica (recebida como parâmetro) nos processos primitivos adequados. Fora a estrutura simbólica, o outro parâmetro deste processo é o alfabeto de eventos da especificação da qual serão selecionados os testes.

```
TP_BUILDER(alfa, tpdata) =
  let
   Start = if(tpdata != <>) then EXTERNAL_BUILDER(tpdata) else RUN(alfa)

   EXTERNAL_BUILDER (<>) = STOP
   EXTERNAL_BUILDER (tps) = INTERNAL_BUILDER( head(tps) )
                            [] EXTERNAL_BUILDER( tail(tps) )

   INTERNAL_BUILDER( (st,p,tps) ) = Start
   INTERNAL_BUILDER( (an,p,tps) ) = ANY(p, EXTERNAL_BUILDER(tps))
   INTERNAL_BUILDER( (no,p,tps) ) = NOT(alfa, p, EXTERNAL_BUILDER(tps))
   INTERNAL_BUILDER( (mt,p,tps) ) = MATCH(alfa, p, EXTERNAL_BUILDER(tps), Start)
   INTERNAL_BUILDER( (ex,p,tps) ) = EXCEPT(alfa, p, EXTERNAL_BUILDER(tps), Start)
   INTERNAL_BUILDER( (un,p,tps) ) = UNTIL(alfa, p, EXTERNAL_BUILDER(tps))
   INTERNAL_BUILDER( (ac,p,tps) ) = ACCEPT(p)
   INTERNAL_BUILDER( (re,p,tps) ) = REFUSE(p)
  within Start
```

Como exemplo, ATG utiliza a representação simbólica de PPT, PPT2 e PPT3 como parâmetros de entrada para TP_BUILDER e define o comportamento dos processos CPPT, CPPT2 e CPPT3.

```
CPT2 = TP_BUILDER(AlfaS,PPT2S)

CPT3 = TP_BUILDER(AlfaS,PPT3S)

CPT = TP_BUILDER(AlfaS,PPTS)
```

Os processos CPPT, CPPT2 e CPPT3 correspondem ao propósitos de teste reais especificados a princípio como os pseudo propósitos PPT, PPT2 e PPT3.

6.2 Ambiente de experimentação

A Figura 6.3 apresenta uma idéia simplificada do fluxo do processo de geração de teste onde foi realizado o estudo de caso. Considerando que os testes gerados estão no nível de sistema (ver Seção 2.1) seguindo a abordagem caixa-preta (ver Seção 2.3) e as especificações de comportamento são descritas em linguagem natural (Inglês). Os casos de teste são especificados e executados de forma manual. Os testes são produzidos pelo projetista de teste a partir da leitura dos requisitos e documentados em linguagem natural. O testador, seguindo a especificação dos casos de teste (e procedimentos de teste), executa os testes. Eventualmente, baseado na especificação dos testes manuais, desenvolvedores escrevem scripts de automação em Java que são utilizados para execução automática dos testes.

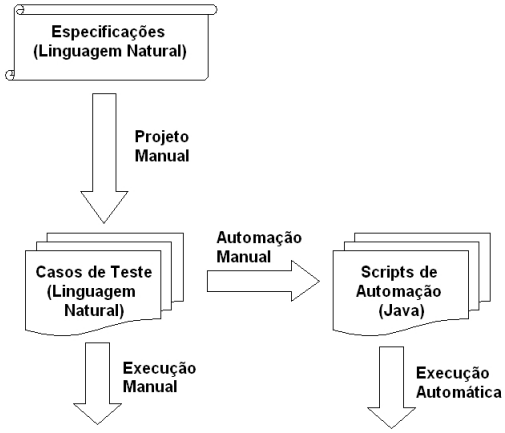

Figura 6.3 Processo do ambiente do estudo de caso

Este processo possui dificuldades clássicas como o alto esforço empregado para produzir os casos de testes (de forma manual). Os requisitos que são a fonte de informação para a especificação dos testes, por estarem em linguagem natural, possibilitam várias interpretações e podem conter redundâncias. Outro problema está relacionado à duplicação de esforço, devido

ao tamanho avantajado dos grupos que criam e mantém os conjuntos de teste. Podem existir vários casos de teste duplicados para testar as mesmas funcionalidades. Ademais, não é garantido que os testes gerados cubram totalmente o escopo dos requisitos de entrada, pois é utilizada intuição humana para decidir a cobertura de cada teste especificado e rodado.

No contexto de pesquisa onde esse trabalho se insere, o CINBTC-RD, a solução proposta parte de requisitos descritos em linguagem natural e chega em casos de teste também em linguagem natural. Entretanto, a abordagem de geração de ATG trabalha com especificações CSP de entrada e produz como saída casos de teste CSP. A ligação entre o universo formal de CSP e a linguagem natural é resolvida pela integração com outras ferramentas produzidas no projeto CINBTC-RD. A Figura 6.4 mostra como a partir do processo original é possível aplicar a nossa abordagem de geração que trabalha com CSP.

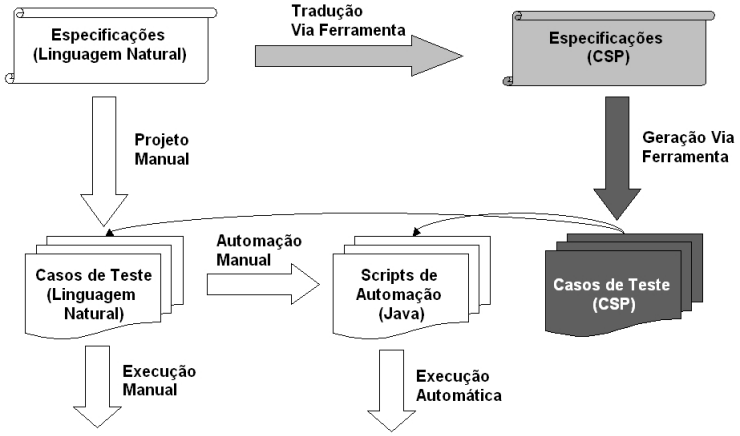

Figura 6.4 Adaptações no processo para estudo de caso

O lado esquerdo da Figura 6.4 consiste no processo original, já o lado direito (destacado pelas cores cinza claro e cinza escuro) consiste nas ferramentas utilizadas para aplicação do estudo. A tradução das especificações em linguagem natural para especificações CSP (elementos em cinza claro) é realizada por uma ferramenta de processamento de linguagem natural (fora do escopo desse trabalho, para mais informações consultar [Tor06] e [dFLC06]). A saída desta ferramenta é uma especificação CSP do comportamento do sistema a ser testado. Tal especificação é entrada para a ferramenta ATG que gera casos de teste CSP (na Figura cinza escuro). Neste ponto do processo, outras ferramentas do projeto (fora do escopo desse trabalho, para mais informações consultar [Lei06]) traduzem os casos de teste CSP para casos de teste em linguagem natural e scripts de automação.

6.3 Aplicação do Estudo

Como o material utilizado para o estudo de caso é propriedade intelectual da Motorola, não será possível fornecer maiores detalhes sobre produtos, processos da empresa, padrão de documentação, etc. Felizmente, a maioria das lições e experiências constituem material não confidencial,

portanto, serão relatadas a seguir.

O objetivo desse estudo foi comparar a quantidade e qualidade dos testes gerados automaticamente a partir de uma especificação formal, com os testes existentes (gerados manualmente) para o uma aplicação e IUT selecionadas. A metodologia seguida é constituída dos seis seguintes passos:

- **Passo 1** – Escolha de uma aplicação (caso) bem especificada e que contenha especificações de casos de teste para execução manual. Escolha de uma IUT (estável) que implemente esta aplicação;
- **Passo 2** – Obtenção da especificação CSP a partir da documentação do comportamento da aplicação;
- **Passo 3** – Definição de pseudo propósitos de teste CSP baseados no alfabeto da especificação CSP definida no passo anterior. E, usar ATG para automaticamente gerar testes a partir desta especificação e destes propósitos de teste; e,
- **Passo 4** – Avaliação da qualidade/quantidade dos testes obtidos automaticamente, comparando com os testes pré-existentes (gerados manualmente) da aplicação.

  As subseções seguintes descrevem em detalhes cada um desses passos.

6.3.1 Escolha da Aplicação

Iniciando pelo **Passo 1** da metologia, a aplicação escolhida para o estudo foi um sistema embarcado para celulares de gerenciamento de mensagens SMS (Short Message Service) [Wik06c]. Esta aplicação foi escolhida por já possuir um conjunto de testes funcionais (sistema e caixa-preta) manualmente produzido e consolidado que pudesse servir de referência para uma comparação com os testes gerados automaticamente por ATG. Outro motivo para a escolha, foi a qualidade da documentação de requisitos existente e documentação dos casos de uso, o que facilitaria (posteriormente) a obtenção da especificação formal a partir dos mesmos requisitos. A plataforma de hardware foi um versão com tecnologia GSM (Global System for Mobile Communications) [Wik06b] considerada estável. A estabilidade foi importante para implantação de uma ferramenta de instrumentação de código utilizada para medir a cobertura estrutural e análise de redundância (para mais informações sobre o processo de instrumentação consultar [Soa06]).

6.3.2 Geração da Especificação Formal

A etapa seguinte do estudo, o **Passo 2**, foi gerar uma especificação formal e definir o processo de comportamento (PC) partindo das especificações de requisitos e casos de uso da aplicação de mensagens escolhida no passo anterior do estudo. Isto foi realizado pela equipe do projeto CINBTC-RD (especializada em linguagem natural) em dois passos. No primeiro passo, a equipe transcreveu manualmente as informações sobre as funcionalidades e comportamentos

encontrados na documentação original da aplicação para um formato de Inglês padronizado[1]. Neste formato, as palavras e regras gramaticais são predefinidas e encontram-se todas armazenadas em uma base de conhecimento que pode ser enriquecida e ampliada. Os requisitos funcionais são descritos em termos de ações do usuário, estados e respostas do sistema, e classificados em fluxos (principal, alternativos e de exceções). Partindo das descrições em Inglês padronizado, no passo seguinte, a ferramenta de processamento de linguagem natural (Figura 6.4 na cor cinza clara) foi empregada para traduzir as referidas descrições para CSP (uma especificação CSP). Desta maneira, cada frase do Inglês padronizado (utilizada para descrever ação do usuário, estado e resposta do sistema) foi convertida em um evento CSP específico.

Uma característica relevante da especificação CSP obtida é que a mesma não lida com variáveis (os processos não contém variáveis). As variações entre os fluxos principal, alternativos e de exceções ocorrem a partir da especificação de diferentes prefixos. Cada partição de equivalência (ver Análise do Valor Limite na Seção 2.3) dos valores de entradas, saídas e estados do sistema encontrados na especificação em Inglês padronizado, se tornou um prefixo diferente da especificação CSP. Devido a esta simplificação, o tamanho do alfabeto da especificação obtido foi mais reduzido e menos expressivo em se comparando com um modelo com variáveis explícitas.

Como a aplicação de mensagens contava com um conjunto grande de testes manuais criados a partir dos requisitos, como também e de informações complementares baseadas na experiência dos testadores e projetistas de teste.

Seria incoerente comparar os testes gerados automaticamente a partir da especificação formal com os testes pré-existentes, pois os últimos continham informações e cenários complementares que não estavam nos requisitos (vinham da experiência do projetista de testes e de outras documentações não oficiais). Claramente, estas informações não seriam encontradas nos testes gerados pela ferramenta, porque a especificação usada como entrada para gerá-los não conheceria tais informações. A saída foi enriquecer a representação em Inglês padronizada dos requisitos com os comportamentos complementares, e obter uma nova especificação CSP que os englobasse.

6.3.3 Definição dos Pseudo Propósitos de Teste e Geração dos Testes

Em seguida, no **Passo 3** do estudo de caso, foi definido um pseudo propósito de teste para cada requisito da aplicação, como também um para cada uma das informações complementares usadas para enriquecer a especificação. A partir do alfabeto de eventos da especificação CSP (gerada no passo anterior do estudo) foram definidos 12 pseudo propósitos de teste (6 baseados nos requisitos e 3 baseados em informações complementares). A definição dos propósitos foi a atividade que mais demandou esforço deste estudo, dada a dificuldade em definí-los manualmente. Uma dificuldade encontrada nesta definição foi o tamanho reduzido do alfabeto de eventos da especificação, que continha em sua maioria eventos de significação muito abstrata, por exemplo, um único evento da especificação é utilizado para expressar a ação enviar mensagem (independente de quantos tipos de mensagem existem). Este tipo de abstração deu

[1]Caso a especificação padrão do processo estivesse no formato de Inglês padronizado, não seria necessário realizar a transcrição manualmente.

origem a propósitos de teste cuja seleção de testes foi menos precisa, pois faltavam eventos mais específicos do alfabeto da especificação capazes de descrever variações de um mesmo comportamento (no exemplo anterior, faltavam eventos distintos para cada tipo de mensagem enviada).

Os 12 pseudo propósitos de teste foram então empregados como entrada de ATG, junto com a especificação, para gerar os casos de teste CSP. Foram gerados, em poucos segundos, 37 casos de teste. É praticamente inviável (para o usuário que desconhece CSP) executar os casos de teste diretamente especificados em CSP, pois o significado do alfabeto de eventos é de difícil compreensão. Para facilitar a leitura e execução deste testes, os mesmo foram traduzidos para uma notação de Inglês padronizado para especificar casos de teste. Esta última tradução foi realizada por um ferramenta de processamento de linguagem natural (que na Figura 6.4 corresponde a seta que leva da caixa Casos de Teste CSP para a caixa Casos de Teste Linguagem Natural).

6.3.4 Avaliação dos Testes

Finalmente, no **Passo 4** do estudo, foi realizada a comparação quantitativa e qualitativa dos conjuntos de testes gerados automaticamente por ATG (obtidos no passo anterior deste estudo) com o conjunto de testes gerados manualmente (pela atividade Projeto Manual representada na Figura 6.3). O primeiro conjunto contém 37 testes contra 84 do segundo. A primeira explicação para essa diferença (47 testes) está relacionada as características da especificação CSP utilizada que estão relacionadas a seguir.

- A especificação possui um nível de abstração muito alto, portanto gera casos de teste com o mesmo nível de abstratação (ver comentário da Seção 6.3.2). A abstração dos eventos da especificação fez com que alguns testes gerados automaticamente pudessem ser relacionados com quatro diferentes testes que foram especificados manualmente. Como exemplo dessa situação, um teste CSP que descreve o fluxo de envio de uma mensagem (independente do tipo) equivale a quatro testes projetados manualmente para o mesmo fluxo. Os testes manuais especificavam o mesmo fluxo do teste CSP, com a diferença de que cada um destes quatro testes enviava uma mensagem de tipo diferente.

- Ainda considerando as abstrações da especificação CSP, um único caso de teste CSP gerado equivalia a vários casos de teste que foram manualmente projetados. Isso ocorre, pois em muitos pontos da especificação CSP é descrito o comportamento comum para toda uma partição de valores (ver Análise do Valor Limite na Seção 2.3). Por exemplo, a especificação CSP descreve o comportamento da visualização do conteúdo de uma mensagem quando existem mais de 2 mensagens armazenadas na caixa de mensagens, portanto, o conjunto de testes CSP gerado contém apenas um teste para essa situação (que testa o comportamento comum para a partição de valores onde existem 3 ou mais mensagens na caixa de entrada). Para o mesmo comportamento anterior, o conjunto de testes manuais possui múltiplos testes : um teste quando a caixa de entrada possui 3 mensagens, outro teste quando possui 4 mensagens e assim por diante.

- Foi observada a tendência geral de que os casos de teste formais eram mais longos e

sofisticados do que os casos de teste gerados manualmente. Um único caso de teste CSP cobria a funcionalidade de vários casos de teste especificados manualmente.

- Uma última explicação para que o conjunto de testes CSP possuisse 100 testes a menos do que o número total de testes do conjunto criado manualmente, é que apesar de enriquecer a especificação CSP com algumas informações complementares (ver Seção 6.3.2), ainda sim os testes manuais eram mais ricos em informações e especificavam situações que não foram acrescentadas na especificação.

A análise qualitativa entre os dois conjuntos de teste considerou: a cobertura estrutural alcançada e o nível de redundância dos testes de cada um dos conjuntos. A análise de cobertura consiste em medir a porcentagem de métodos e classes do código-fonte da implementação que foram percorridos durante a execução de um conjunto de testes sobre esta IUT. Quanto maior a porcentagem dos elementos da implementação forem cobertos melhor será a cobertura. Quando existem elementos da implementação que são cobertos por mais de um teste no mesmo conjunto então diz-se que ocorre redundância. As características desejadas para um conjunto de testes são alta taxa cobertura e baixa taxa de redundância.

Devido a limitações de recurso para executar 121 casos de teste (84 gerados manualmente e 37 gerados automaticamente) foi escolhido um subconjunto de 31 testes (26 gerados manualmente e 5 gerados automaticamente) relativo a um dos casos de uso da aplicação de gerenciamento de mensagens. Como nos 26 casos de teste gerados manualmente existiam aspectos não contemplados na especificação formal (pois os testes manuais são mais ricos do que os documentos nos quais a especificação CSP foi criada), foi selecionado um subconjunto de 11 teste (6 testes funcionalmente equivalentes aos 5 gerados automaticamente para o mesmo caso de uso). Foi observado que para algumas partes do código-fonte a implementação foi melhor pelos testes automático e noutras melhor coberta pelos testes manuais, a análise de redundância também observou essa variação. Na média, para o subconjunto de 11 testes, a cobertura e a redundância dos testes gerados manualmente e automaticamente é igual.

Para mais informações sobre a análise qualitativa consultar [Soa06].

Capítulo 7

Conclusão

Este trabalho introduz uma abordagem de geração automática de casos de testes consistentes (*sound*) guiada por propósitos de teste, baseada na álgebra de processos CSP. A especificação de entrada (modelo), a definição de propósitos de teste e os casos de teste gerados utilizam a notação, os operadores e ferramentas de CSP. Esta abordagem não necessita manipular a semântica operacional (LTS) de CSP explicitamente, sendo, portanto, uma alternativa às abordagens tradicionais que se baseiam em formalismos mais operacionais.

Como todos os elementos da especificação são processos CSP, eles podem ser combinados para constituir novos elementos: especificações, propósitos de teste e casos de teste podem ser combinados usando os operadores de CSP. Esta facilidade de composição torna a evolução da especificação (modelo) potencialmente mais natural. Além disso, tal abordagem pode facilitar o emprego de abordagens de engenharia reversa de especificações [BPIM04] (*Anti-Model-Based*) que combinam casos de teste para obter novas especificações. Em particular, esta característica é extremamente relevante no contexto onde este projeto foi desenvolvido: o projeto de pesquisa do Brazil Teste Center (BTC), uma cooperação entre o CIn-UFPE e a Motorola Industrial Limitada.

Propósitos de testes CSP são processos formados pela combinação de vários processos primitivos, definidos de forma a permitir a especificação de cenários de teste de forma livre e abstrata. Por cenários, entende-se as propriedades definidas a partir de padrões de ocorrência de eventos que se deseja encontrar na especificação. Quando um cenário está presente na especificação, então existe um *trace* da especificação onde a propriedade especificada poderá ser encontrada; tal trace é chamado de cenário de teste. Podem ser especificados cenários de aceitação (que posteriormente vão originar casos de teste) como também cenários de refutação (que não interessam e são cortados da seleção). Propósitos de teste CSP, quando combinados, formam novos propósitos cuja capacidade de especificação de cenários de teste torna-se mais poderosa.

Baseado na descrição de uma propriedade a ser testada, na forma de um propósito de testes CSP, verificações de refinamento entre processos (que usam o modelo de *traces*) são empregadas e retornam como contra-exemplos os cenários de teste (*traces*) onde a propriedade especificada pode ser encontrada. Pode ser obtido desde o menor cenário de teste onde um propósito de testes é válido, até a quantidade máxima existente (caso seja finita). Caso nenhum contra-exemplo seja retornado, então o cenário especificado pelo propósito de teste não está presente no comportamento do processo de especificação; neste caso, o propósito é inconsistente, isto é, não é capaz de originar cenários de teste. Os cenários de teste selecionados via propósitos de teste CSP são utilizados para construir casos de teste coerentes.

Uma relação de conformidade denotada **cspioco** é apresentada em termos de refinamento

de processos para definir se o processo que representa a implementação a ser testada está em conformidade com o comportamento do processo da especificação. Tal relação assume como hipótese que a implementação a ser testada possa ser modelada através de um processo CSP, cujo alfabeto de eventos está particionado em entradas e saídas.

A partir da relação **cspioco** é formulada uma teoria de testes onde um caso de teste CSP é representado por um processo que interage com um implementação (de comportamento arbitrário) para aferir se a mesma está conforme ou não a especificação, isto é, se a implementação satisfaz a relação **cspioco** com relação à especificação. Tal caso de teste é executado contra a IUT de acordo com um modelo de execução apresentado. Via refinamentos e internalização de eventos, é demonstrada a forma de detectar os eventuais vereditos alcançados na execução de um caso teste CSP. Partindo destas definições, é apresentada (em termos de refinamentos de processo) a propriedade de consistência (*soundness*) para um caso de teste CSP, de acordo como a relação de conformidade **cspioco** .

A estrutura fundamental de um caso de teste CSP corresponde ao comportamento do processo *TP_BUILDER*, quando este recebe como parâmetro um cenário de teste previamente selecionado. A partir desta estrutura inicial, o caso de teste é refinado iterativamente até alcançar uma definição que seja consistente com a especificação. O processo final representa um caso de teste CSP que verifica se a implementação testada apresenta o cenário de teste fornecido: passa quando o cenário de testes fornecido é encontrado, é inconclusivo quando a implementação apresenta comportamentos especificados diferentes do cenário de teste fornecido, e falha se a implementação apresenta um comportamento que não está especificado.

A automação da abordagem de geração de testes proposta é implementada pela ferramenta ATG. Esta ferramenta foi implementada em Java e interage com FDR que realiza as verificações de refinamentos e retorna contra-exemplos que são processados pela ferramenta. ATG aceita como entrada um formato de descrição simplificado para definição de propósitos de teste CSP, denominado pseudo propósito de teste. A partir de uma especificação CSP e dos pseudo propósitos de teste, ATG verifica a consistência dos pseudo propósitos de teste e também gera os casos de teste. Fora estas funcionalidades, ATG também permite: verificar as propriedades clássicas de processos (ausência de *deadlock*, ausência de *livelock* e comportamento determinístico), verificar conformidade dos casos de teste (se um conjunto de testes está de acordo com uma especificação dada) e gerar o LTS de um processo CSP (no formato que pode ser manipulado por uma ferramenta de edição de grafos).

Finalizando o trabalho, foi desenvolvido um estudo de caso real realizado dentro do ambiente do projeto de testes BTC. Tal estudo objetivou comparar os testes gerados manualmente (no nível de sistema, de acordo com a abordagem caixa-preta) a partir de especificações em linguagem natural (inglês), com os testes CSP gerados automaticamente por ATG. Foram definidos pseudo propósitos de teste a partir dos requisitos da aplicação e gerados os testes CSP que foram convertidos (por outras ferramentas do projeto de pesquisa CINBTC-RD) para um formato padrão de inglês para casos de teste. Em quantidade, foram gerados muito menos testes automáticos do que os testes manuais. Isto pode ser compreendido pelas várias simplificações e abstrações inerentes ao modelo formal utilizado por ATG para gerar os testes. Apesar das simplificações na especificação CSP, comparando a cobertura e a redundância entre os dois grupos de teste (manuais e automáticos), foi verificado que, para um mesmo caso de uso, testes

que são funcionalmente equivalentes, na média, possuem a mesma cobertura e nível de redundância estrutural, sejam eles automáticos ou manuais. Outro fator bastante positivo do estudo foi a redução de esforço e a facilidade observada para gerar os casos automaticamente em comparação aos testes manuais. A geração automática de testes guiada por propósitos CSP permite realizar uma seleção dos testes que é relacionada diretamente com os requisitos especificados pelos propósitos de teste fornecidos como entrada.

7.1 Trabalhos Relacionados

Em [RW05], considerando um processo CSP que especifica o comportamento de diagramas de seqüência UML, CSP e FDR são utilizados para validar a presença ou ausência de cenários (seqüências de eventos fornecidas) no comportamento da especificação. Expressões de refinamento entre processos (utilizando modelo de traces, de falhas e divergências) são empregadas para verificar propriedades de *liveness* (se dado cenário eventualmente ocorre na especificação) e *safety* (se dado cenário sempre ocorre na especificação). Entretanto, nenhum dos trabalhos anteriores, que se tem conhecimento, utiliza CSP e FDR como únicas ferramentas para geração de testes, seja dirigida ou não por propósitos de teste.

Técnicas de geração de teste peculiares a outros formalismos empregam propósitos de testes para descrever propriedades particulares do sistema a ser testado como em [Soc06], que utiliza MSC, ou [JJ05], que utiliza IOLTS, ou como [Hol04], que emprega Lógica Temporal. Os dois primeiros formalismos são usados geralmente para especificar propriedades de *liveness*, enquanto lógica temporal especifica tanto propriedades de *safety* quanto propriedades de *liveness*. Propósitos de teste CSP especificam propriedades de *liveness*.

A técnica que utiliza refinamento de *traces* entre processos apresentada neste trabalho lembra o princípio utilizado para obtenção de seqüência de teste a partir de propriedades em Lógica Temporal. Enquanto nossa abordagem utiliza refinamentos para obtenção de contra-exemplos (cenários de teste) da especificação, que possuem as propriedades especificadas pelo propósito de teste CSP, técnicas de Lógica Temporal empregam a verificação da negação de uma propriedade temporal (descrita como uma fórmula temporal) para obter *traces* de contra-exemplos onde aquela propriedade é verdade na especificação. Enquanto as fórmulas da Lógica Temporal descrevem as propriedades dos testes a serem obtidos em termos do valor das variáveis encontradas nos estados da especificação, propósitos de teste CSP descrevem propriedades (cenários) com respeito à ocorrência de eventos na especificação.

7.1.1 TGV

Nossa abordagem baseia-se na relação de implementação **cspioco** , cuja hipótese é que a IUT possa ser modelada por um processo CSP. TGV baseia-se na relação **ioco** e na Teoria de Testes de Tretmans [Tre96b], projetada especificamente para testar sistemas reativos (onde a IUT nunca rejeita as entradas fornecidas pelo teste), cuja hipótese é que a IUT possa ser modelada via um IOLTS. Portanto, claramente existem alguns pontos correlatos entre TGV e nossa abordagem, no que diz respeito às teorias utilizadas.

- O modelo de execução de um teste com a IUT em nossa abordagem é montado como a composição paralela do processo que representa a IUT com o processo do caso de teste. Neste modelo de execução, a comunicação entre teste e implementação é síncrona: saídas do teste sincronizam nas saídas do sistema e as entradas do teste nas entradas do sistema. Os vereditos da execução são alcançados quando o teste comunica um dos eventos de veredito, o que provoca um *deadlock* que sinaliza o fim da execução do teste. Teste e implementação podem ocasionalmente entrar em deadlock em alguns pontos onde nenhum veredito está definido. Já em TGV, a execução é entendida como a execução paralela de dois IOLTS (IUT e teste), onde as saídas do caso de teste sincronizam com as entradas da implementação, e as saídas da implementação sincronizam com as entradas do caso de teste. Sempre que ocorrer um *deadlock* na execução do teste, um estado de veredito do teste é alcançado; e,

- Os vereditos que podem ser alcançados por um caso de teste CSP em nossa teoria são: passou, falhou e inconclusivo. Os vereditos da teoria de teste de Tretmans são os mesmos, com excessão do significado para inconclusivo, que é diferente. Na nossa teoria, o veredito inconclusivo acontece quando, durante a execução do teste, a implementação apresenta um comportamento especificado, porém diferente do comportamento do cenário de teste. Na Teoria de Tretmans, inconclusivo ocorre quando se especifica, no propósito de teste, um comportamento que deve ser desconsiderado da seleção e, durante a execução do teste contra a IUT, a IUT se comporta de acordo com o comportamento a desconsiderado.

Também existem vários pontos correlatos nas técnicas utilizadas para obter testes CSP, a partir de propósitos de teste CSP, e das técnicas empregadas para geração de testes IOLTS, a partir de propósitos IOLTS utilizado por TGV [JJ05] :

- A especificação CSP possui uma separação sintática dos alfabetos de entrada e saída (dois conjuntos são definidos para especificar tal separação); já IOLTS em TGV possui uma separação semântica dos eventos de entrada e saída (faz parte da definição de IOLTS);

- Os propósitos de teste CSP são determinísticos e completos (aceitam todos os eventos do alfabeto da especificação em qualquer momento) e os propósitos IOLTS também o são;

- Na medida em que propósitos de teste CSP são equipados com eventos de demarcação para aceitação (canal *accept*) e recusa (canal *refuse*), TGV emprega estados de aprisionamento (*trapstates*) para aceitação e recusa; e,

- Enquanto nossa abordagem monta uma expressão com processos CSP (chamada de produto paralelo), cujo comportamento corresponde à expansão do paralelismo síncrono (entre o processo que representa a especificação com o processo do propósito de teste CSP) para demarcar as porções da especificação CSP onde a propriedade especificada pelo propósito pode ser encontrada, TGV utiliza o algoritmo de produto síncrono (que multiplica o comportamento do IOLTS da especificação com o IOLTS do propósito de teste) para este mesmo fim;

As etapas posteriores para obter casos de teste consistentes (*sound*) diferem bastante em finalidade e complexidade, comparando à nossa abordagem que usa CSP com as técnicas de TGV, pois nossos testes estão de acordo com a relação de implementação **cspioco** , enquanto os testes de TGV estão de acordo com a relação **ioco** . Nossa abordagem utiliza o processo parametrizado *TC_BUILDER*, que recebe como parâmetro o cenário de teste, e, após sucessivas verificações de refinamento e ajustes dos valores de entrada de *TC_BUILDER*, produz uma versão do caso de teste que é consistente. Já TGV utiliza o grafo resultante do produto síncrono para realizar várias transformações: torna-o determinístico, resolvendo conflitos, invertendo eventos de entrada em eventos de saída e vice-versa, até alcançar um caso de teste IOLTS. É indiscutível que a abordagem de geração CSP é bem mais simples do que a utilizada por TGV, apesar das finalidades serem distintas.

Propósitos de teste CSP são definidos através de processos primitivos que possuem uma facilidade muito superior para descrever propriedades do que as transições e estados dos propósitos IOLTS. Esta facilidade se deve ao fato de que os primeiros possuem um nível maior de abstração e modularidade, pois são formados por processos mais simples que podem ser combinados para formar novos propósitos mais expressivos. Em contrapartida, a notação dos propósitos IOLTS possui pouca abstração, pois os elementos da notação são basicamente transições, estados e algumas expressões regulares. IOLTS também não possuem modularização (formam um bloco monolítico) pois não possuem operadores de composição.

7.2 Trabalhos Futuros

Apesar de que a teoria de testes e a abordagem de geração propostas empregam elementos específicos de CSP, acreditamos que várias das técnicas propostas podem ser empregadas de forma similar em outras álgebras de processos como CCS[Mil89] e LOTOS [psI89]. Fica como trabalho futuro verificar a viabilidade desta teoria e da abordagem de geração com estes e outros formalismos.

Uma implementação é **cspioco** com relação a uma especificação, se a implementação possui uma quantidade menor ou igual de saídas que podem ser encontradas na implementação, considerando os *traces* da especificação. Caso a implementação em questão possua mais situações de *deadlocks* e *livelocks* que a especificação, considerando os traces comuns entre ambas, de acordo com **cspioco** será uma implementação válida. Uma extensão deste trabalho é elaborar uma nova relação de conformidade (também em termos de refinamentos de processos) que considere que os comportamentos de *deadlock* e *livelock* contidos na implementação devem existir em menor ou igual quantidade aos da especificação, considerando os *traces* comuns entre ambas. O modelo de falhas e divergências pode ser utilizado para elaboração desta nova relação de implementação que pode incluir as verificações de **cspioco** (que utiliza a semântica de *traces*) e em acréscimo, leve em conta *deadlocks* e *livelocks*.

Alguns processos primitivos interessantes para elaboração de propósitos de teste foram propostos, entretanto, como trabalho futuro, outros processos que sejam mais elaborados e expressivos podem ser definidos a partir da extensão ou reuso dos existentes.

Como, em muitas situações, a escolha externa entre propósitos de teste primitivos (ou não) faz o propósito de teste resultante ter um comportamento não-determinístico (uma propriedade

indesejável), é necessário reescrever esta combinação de forma que seja mantido o comportamento determinístico. No momento, esta reescrita é feita de forma intuitiva e manual, mas a partir do momento em que regras sejam definidas para as diferentes combinações entre os processos primitivos, isto pode ser realizado de forma automática via suporte de ferramenta. Fica como contribuição futura definir tais regras.

Dado o alto esforço investido para elaborar os pseudo propósitos, pois esta atividade envolve a exploração do comportamento e do alfabeto da especificação, contribuições futuras na direção de automatizar ou semi-automatizar esta atividade serão muito bem vindas. Como sugestão, uma ferramenta poderia reduzir esforço desta atividade permitindo a escrita dos propósitos de teste em um padrão de linguagem natural controlada que fosse traduzido automaticamente para o formato dos propósitos de teste CSP. Tal ferramenta poderia completar a especificação do propósito de testes à medida que a mesma fosse sendo escrita pelo usuário (oferecendo os eventos do alfabeto descritos em linguagem natural), ao mesmo tempo que também poderia validar a especificação.

Correntemente, o processo de produção de casos de teste necessita de vários ajustes consecutivos (várias iterações de verificações de refinamento) para tornar um caso de teste coerente. Um trabalho futuro seria reduzir a quantidade de iterações necessárias para ajustar o caso de teste, ou mesmo já construí-lo coerente (sem necessidade de ajustes).

APÊNDICE A

Exemplos CSP_M

Este apêndice lista as especificações CSP_M correspondentes aos exemplos utilizados no Capítulo 5 para ilustrar o processo de geração de cenários de teste baseados em propósitos de teste CSP.

A.1 Especificação da Seção 5.1

Especificação CSP_M para o processo *SYSTEM*.

```
AlfaSi = {a,b,c}
AlfaSo = {x,y,z}
AlfaS = union(AlfaSi,AlfaSo)

channel a,b,c,x,y,z
channel t

SYSTEM = let
S0 = t -> S9 [] t -> S2
S2 = t -> S0 [] (c -> S6 [] b -> S4)
S4 = z -> S2 [] t -> S4 [] t -> S8
S6 = y -> S7
S7 = c -> S6
S8 = y -> S0
S9 = a -> S8
within S0\{t}

S = SYSTEM

assert S :[ deadlockfree [F] ]
assert S :[ divergence free ]
assert S :[ deterministic [FD] ]
```

A.2 Especificações da Seção 5.2

CSP_M para o processo $MATCH_{seq}$.

```
channel match

MATCHSEQ(alfa,<>,start) = match -> STOP
MATCHSEQ(alfa,s,start) =
      head(s) -> MATCHSEQ(alfa,tail(s),start)
      []
      ([] ev : diff(alfa,{head(s)}) @ ev -> start)
```

CSP_M para o exemplo (5.4).

```
seq1 = <a,y>

MATCH1 = MATCHSEQ(AlfaS,seq1,MATCH1)

-- refinamento é válido, pois 'seq1' é um seqüência que eventualmente
-- pode ser encontrada em um cenário de SYSTEM.
assert ( SYSTEM [|AlfaS|] MATCH1 ) \ AlfaS [T= match -> STOP
```

CSP$_M$ para o exemplo (5.5).

```
seq2 = <a,b>

MATCH2 = MATCHSEQ(AlfaS,seq2,MATCH2)

-- refinamento é inválido, pois 'seq1' é um seqüência que eventualmente
-- não pode ser encontrada em um cenário de SYSTEM.
assert ( SYSTEM [|AlfaS|] MATCH2 ) \ AlfaS [T= match -> STOP
```

A.3 Especificações da Seção 5.3

Especificação *CSP*$_M$ para a seleção do cenário de teste onde eventualmente ocorre *seq*.

```
-- refinamento inválido que retorna o contra-exemplo <a,y,match>
assert SYSTEM [T= SYSTEM [|AlfaS|] MATCH1
```

A.4 Expecificações da Seção 5.3.1

Especificações *CSP*$_M$ para obtenção de múltiplos cenários de teste onde eventualmente ocorre a propriedade *seq*.

```
PREFIX(<>) = STOP
PREFIX(seq) = head(seq) -> PREFIX(tail(seq))

CE1 = <a,y,match>

-- refinamento inválido que retorna CE2 como contra-exemplo
assert SYSTEM [] PREFIX(CE1) [T= SYSTEM [|AlfaS|] MATCH1

CE2 = <b,y,a,y,match>

-- refinamento inválido que retorna CE3 como contra-exemplo
assert SYSTEM [] PREFIX(CE1) [] PREFIX(CE2) [T= SYSTEM [|AlfaS|] MATCH1

CE3 = <b,z,a,y,match>

-- refinamento inválido que retorna CE4 como contra-exemplo
assert SYSTEM [] PREFIX(CE1) [] PREFIX(CE2) [] PREFIX(CE3) [T= SYSTEM [|AlfaS|] MATCH1

CE4 = <b,z,b,y,a,y,match>
```

A.5 Especificações da Seção 5.4

Especificação *CSP*$_M$ para os processos básicos para construção de PTCSPs.

```
MAX_MARKS = 10

channel accept,refuse : {1..MAX_MARKS}

ACCEPT({n}) = accept.n -> STOP

REFUSE({n}) = refuse.n -> STOP

RUN(evs) = [] ev : evs @ ev -> RUN(evs)

ANY(evs,next) = [] ev : evs @ ev -> next

NOT(alfa, evs,next) = ANY(diff(alfa,evs), next)

MATCH'(alfa, evs, next, initial) = ANY(evs, next) [] NOT(alfa, evs, initial)

EXCEPT(alfa, evs, next, initial) = MATCH'(alfa,evs,initial,next)

UNTIL(alfa, evs, next) = RUN(diff(alfa,evs)) /\ ANY(evs,next)
```

Especificação CSP_M da Figura **??** aplicada a seqüência *seq*1.

```
channel null
element(index,<>) = null
element(1,s) = head(s)
element(index,s) = element(index-1,tail(s))

ANY_seq1 = MATCH'(AlfaS,{element(1,seq1)},TP1,ANY_1)
ANY_1 = MATCH'(AlfaS,{element(2,seq1)},TP1',match -> STOP)

-- refinamento inválido que retorna o contra-exemplo <a,y,match>
assert SYSTEM [T= SYSTEM [|AlfaS|] ANY_seq1
```

Especificação CSP_M da Seção 5.4.2 .

```
PT = ANY( {y}, PT')
     []
     ANY( {z}, REFUSE({1}))
     []
     NOT( AlfaS, {y,z} , PT)

PT'  = ANY ( {z}, ACCEPT({1}))
       []
       NOT (AlfaS, {z}, PT')

-- produto paralelo entre SYSTEM e PT
PP = SYSTEM [|AlfaS|] PT

-- verificação de comportamento determinístico válida
assert PP :[deterministic F]

-- refinamento inválido que retorna o contra-exemplo CE1'
assert SYSTEM [T= PP

CE1' = <b,z,refuse.1>

-- refinamento inválido que retorna o contra-exemplo CE2'
assert SYSTEM [] PREFIX(CE1') [T= PP

CE2' = <a,y,b,z,accept.1>
```

Especificação CSP_M da Seção 5.4.3.

```
PT2 = UNTIL(AlfaS,{y},UNTIL(AlfaS,{z},ACCEPT({1})))

PT3 = UNTIL(AlfaS,{z},REFUSE({1}))

-- refinamento inválido que retorna o contra-exemplo CE1''
assert SYSTEM [T= SYSTEM [|AlfaS|] PT2

CE1'' = <a,y,b,z,accept.1>

-- refinamento inválido que retorna o contra-exemplo CE2''
assert SYSTEM [T= SYSTEM [|AlfaS|] PT3

CE2'' = <b,z,refuse.1>

PT4 = UNTIL(AlfaS,{y},UNTIL(AlfaS,{z},ACCEPT({1})))
      []
      UNTIL(AlfaS,{z},REFUSE({1}))

-- verificação de determinismo falha (PT4 é não determinístico)
assert PT4 :[deterministic F]

-- produto paralelo de SYSTEM com PT4
PP2 = SYSTEM [|AlfaS|] PT4

-- refinamento inválido que retorna o contra-exemplo CE1''
assert SYSTEM [T= PP2

CE1''' = <b,z,refuse.1>

-- refinamento inválido que retorna o contra-exemplo CE2''
assert SYSTEM [] PREFIX(CE1') [T= PP2

-- contra-exemplo inconsistente
CE2''' = <a,y,b,z,refuse.1>

-- propósito que seleciona cenários onde eventualmente 'y' e logo em seguida ocorre c
PT5 = UNTIL(AlfaS,{y},MATCH'(AlfaS,{c},ACCEPT({2}),PT5))

-- produto paralelo para seleção de cenários em SYSTEM pelo propósito PT5
PP3 = SYSTEM [|AlfaS|] PT5

-- verificação de refinamento inválida que retorna o contra-exemplo CE1''''
assert SYSTEM [T= PP3

CE1'''' = <c,y,c,accept.2>

-- os propósitos PT6 possuem comportamento equivalente a PT7
PT6 = MATCH'(AlfaS,{z},ACCEPT({2}), PT6)
PT7 = UNTIL(AlfaS,{z},ACCEPT({2}))

-- verificação de equivalência entre processos, utiliza o modelo de falhas
-- e divergências [FD] de CSP. Os dois refinamentos são válidos.
assert PT6 [FD= PT7
assert PT7 [FD= PT6

-- captura cenários onde se espera eventualmete 'y', e imediatamente
-- qualquer evento diferente de 'c'
PT8 = UNTIL(AlfaS,{y},EXCEPT(AlfaS,{c},ACCEPT({3}),PT8))

-- produto paralelo entre SYSTEM e PT8
PP4 = SYSTEM [|AlfaS|] PT8

-- verificação de refinamento inválida que retorna o contra-exemplo CE1'''''
assert SYSTEM [T= PP8

CE1''''' = <a,y,a,accept.3>
```

Especificação CSP_M para o processo $MATCH_S$ da Seção 5.4.4.

```
MATCHS(alfa, <>, next, initial) = next
MATCHS(alfa, s, next, initial) =
       ANY({head(s)},MATCHS(alfa, tail(s), next, initial))
       []
       NOT(alfa, {head(s)}, initial)


-- propósito que seleciona cenários onde eventualmente ocorre 'y', e
-- imediatamente em seguida ocorre a seqüência <a,y>
PT9 = UNTIL(AlfaS,{y},MATCHS(AlfaS,seq1,ACCEPT({4}),PT9))

-- produto paralelo entre SYSTEM e PT9
PP5 = SYSTEM [|AlfaS|] PT9

-- verificação de refinamento inválida que retorna o contra-exemplo CE1''''''
assert SYSTEM [T= PP5

CE1'''''' = <a,y,a,y,accept.4>
```

A.6 Especificações da Seção 5.5

Especificação CSP_M para o processo $IUT1$.

```
channel d

AlfaIi1 = {a,b,c,d}
AlfaIo1 = {x,y,z}
AlfaI1 = union(AlfaIi1,AlfaIo1)

IUT1 =
let
S0 = S9 [] S2 [] S10
S2 = c -> S6 [] b -> S4
S4 = z -> S2 [] S8
S6 = y -> S7
S7 = c -> S6
S8 = y -> S0
S9 = a -> S8
S10 = d -> y -> S4
within S0

-- refinamento é valido pois IUT1 é uma implementação de SYSTEM de acordo
-- com a relação CSPIOCO
assert SYSTEM [T= (SYSTEM ||| RUN(AlfaIo1)) [|AlfaI1|] IUT1
```

Especificação CSP_M para o processo $IUT2$.

```
cAlfaIi2 = {a,b,c}
AlfaIo2 = {x,y,z}
AlfaI2 = union(AlfaIi2,AlfaIo2)

IUT2 =
let
S0 = S9 [] S2
S2 = c -> S6 [] b -> S4
S4 = z -> S2 [] S8
```

```
S6 = y -> S7
S7 = c -> S6
S8 = y -> S0 [] x -> S0
S9 = a -> S8
within S0

-- refinamento é inválido pois IUT2 não é uma implementação de SYSTEM de acordo
-- com a relação CSPIOCO
assert SYSTEM [T= (SYSTEM ||| RUN(AlfaIo2)) [|AlfaI2|] IUT2
```

A.7 Especificações da Seção 5.6.2

Especificação CSP_M para o processo *TC_BUILDER*.

```
channel pass, fail, inconclusive -- not in Alfa

FAIL = fail -> STOP
PASS = pass -> STOP
INCO = inconclusive -> STOP

TC_BUILDER(ialfa,oalfa,<(accept.i,{})>) = PASS

TC_BUILDER(ialfa,oalfa,s) = SUBTC(ialfa,oalfa,head(s)) ; TC_BUILDER(ialfa,oalfa,tail(s))

SUBTC (ia,oa,(ev,acept)) =
if(member(ev,ia)) then
(
ev -> SKIP
)
else
(
ev -> SKIP
[]
ANY(diff(acept,{ev}), INCO)
[]
NOT(union(ia,oa),union(acept,{ev}), FAIL)
)
```

Especificação CSP_M da seqüência s_{CE1}, o caso de teste *CT* obtido a partir de s_{CE1}, o modelo de execução *EXEC* e verificações dos eventuais vereditos desse último processo.

```
-- considerando o cenário CE2' = <a,y,b,z,accept.1>, retornado pela seleção do
-- propósito PT sobre a especificação SYSTEM
sCE1 = <(a,{}),(y,{}),(b,{}),(z,{}),(accept.1,{})>

CT = TC_BUILDER(AlfaSi,AlfaSo,sCE1)

EXEC = SYSTEM [|AlfaS|] CT

-- refinamento válido
assert EXEC\AlfaS [T= pass -> STOP

-- refinamento válido, quando deveria ser inválido, pois o caso de teste não
-- pode falhar em implementações corretas (nesse caso a própria especificação)
assert EXEC\AlfaS [T= fail -> STOP
```

A.8 Especificações da Seção 5.6.3

Especificações CSP_M para obtenção de s'_{CE1}.

```
-- refinamento inválido que retorna o contra-exemplo <a,y,b,y,fail,refuse.1>
assert SYSTEM [|AlfaS|] CT [T= SYSTEM [|AlfaS|] CT [|union(AlfaS,{fail})|] UNTIL(AlfaS,{fail},REFUSE({1}))

-- a partir do contra-exemplo <a,y,b,y,fail,refuse.1> tem-se
sCE1' = <(a,{}),(y,{}),(b,{}),(z,{y}),(accept.1,{})>

-- nova versão para o caso de teste CT
CT' = TC_BUILDER(AlfaSi,AlfaSo,sCE1')

-- novo modelo de execução a partir de CT'
EXEC' = SYSTEM [|AlfaS|] CT'

-- verificação de refinamento válida
assert EXEC'\AlfaS [T= pass -> STOP

-- verificação de refinamento inválida
assert EXEC'\AlfaS [T= fail -> STOP
```

Especificação CSP_M para a verificação dos eventuais vereditos encontrados quando se executa CT' contra a implementação $IUT1$.

```
EXEC_IUT1 = IUT1 [|AlfaI1|] CT'

-- refinamento válido
assert EXEC_IUT1\AlfaI1 [T= pass -> STOP
-- refinamento inválido
assert EXEC_IUT1\AlfaI1 [T= fail -> STOP
```

Especificação CSP_M para a verificação dos eventuais vereditos encontrados quando se executa CT' contra a implementação $IUT2$.

```
EXEC_IUT2 = IUT2 [|AlfaI2|] CT'

-- refinamento válido
assert EXEC_IUT2\AlfaI1 [T= pass -> STOP
-- refinamento inválido
assert EXEC_IUT2\AlfaI1 [T= fail -> STOP
```

APÊNDICE B

Resumo das Primitivas de PTCSP

A listagem abaixo consiste em uma revisão sobre o funcionamento de cada um dos processos apresentados na Seção 5.4 empregados para construir propósitos de teste CSP. Lembrando que cada um desses processos é utilizado no lado direito da expressão do produto paralelo (5.8). Seja Σ_P o alfabeto do processo da especificação que está a esquerda do paralelismo. Então:

ACCEPT(*i*) Demarca o produto paralelo com o evento *accept*.*i* e paraliza a expansão do paralelismo. Usado para assinalar os pontos interessantes da especificação úteis para extrair cenários de teste;

REFUSE(*i*) Demarca o produto paralelo com o evento *refuset*.*i* e paraliza a expansão do paralelismo. Usado para assinalar os pontos a serem excluídos (não relevantes) da seleção de cenários de teste;

ANY(*evset*, *next*) Sincroniza com qualquer próximo evento *ev* que pertençe ao conjunto *evset* oferecido pela especificação e se comporta como *next*;

NOT(Σ_P, *evset*, *next*) Corresponde ao complemento de *ANY*. Sincroniza com qualquer próximo evento *ev* que pertençe ao conjunto $\Sigma_P - evset$ oferecido pela especificação e se comporta como *next*;

MATCH(Σ_P, *evset*, *next*, *initial*) Sincroniza com o próximo evento do conjunto *evset* oferecido pela especificação e se comporta como o processo *next*, caso contrário, sincroniza como outros eventos oferecidos pela especificação $\Sigma_P - evset$ e volta a se comportar como o processo *initial*;

EXCEPT(Σ_P, *evset*, *next*, *initial*) Possui comportamento inverso de *MATCH*. Sincroniza com o próximo evento do conjunto $\Sigma_P - evset$ oferecido pela especificação e se comporta como o processo *next*, caso contrário, sincroniza como outros eventos *evset* oferecidos pela especificação e volta a se comportar como o processo *initial*;

UNTIL(Σ_P, *evset*, *next*) Sincroniza com qualquer evento *ev* que pertençe a $\Sigma_P - evset$ indefinidamete até que eventualmente sincroniza com um evento que pertençe a *ev* e se comporta como o processo *next*; e,

MATCHS(Σ_P, *seq*, *next*, *initial*) Sincroniza *seq* com a seqüência de eventos oferecida pela especificação e se comporta como *next*, caso contrário, se não for possível sincronizar um a um dos eventos de *seq* do primeiro ao último se comporta como o processo *initial*.

Referências Bibliográficas

[AEE+02] Cimatti A., Clarke E., Giunchiglia E., Giunchiglia F., Pistore M., Roveri M., Sebastiani R., and Tacchella A. Nusmv2: an open source tool for symbolic model checking. In Brinksma E. and Guldstrand Larsen K., editors, *14th international conference on computer aided verification (CAV'02)*, Copenhagen, 27-31 July 2002 2002.

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[Apa06a] Apache. Jmeter - apache jmeter. http://jakarta.apache.org/jmeter/, August 2006.

[Apa06b] Apache Jakarta Project, http://jakarta.apache.org/jmeter/usermanual/index.html. *JMeter - User's Manual*, July 2006.

[BBC+02] J. Bowen, K. Bogdanov, J. Clark, M. Harman, R. M. Hierons, and P. Krause. Fortest: formal methods and testing. *In 26th IEEE Computer Software and Applications (COMPSAC 2002)*, pages 91–101, 2002.

[BBC+03] K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, R. M. Hierons, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Working together: Formal methods and testing. December 2003.

[BBJ+02] Paul Baker, Paul Bristow, Clive Jervis, David King, and Bill Mitchell. Automatic generation of conformance tests from message sequence charts. *Lecture Notes in Computer Science*, 2599 / 2003(0302-9743):24–26, June 2002.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131, 1988.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for ctl. In *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, page 388, Washington, DC, USA, 1995. IEEE Computer Society.

[Bei95] Boris Beizer. *Black-Box Testing*. John Wiley & Sons, 1995.

[ben99] *A feasibility study in formal coverage driven test generation*, 36th Design Automation Conference (DAC 99), June 1999.

[Ber91] Gilles Bernot. Testing against formal specifications: a theoretical view. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Advances in distributed computing (ADC) and colloquium on combining paradigms for software development (CCPSD): Vol. 2*, pages 99–119, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[BGM] Marius Bozga, Susanne Graf, and Laurent Mounier. If-2.0: A validation environment for component-based real-time systems.

[BH89] F. Belina and D. Hogrefe. The ccitt-specification and description language sdl. *Comput. Netw. ISDN Syst.*, 16(4):311–341, 1989.

[BJ78] Dines Bjorner and Clifford Jones. *The Vienna Development Method: The Meta-Language*, volume 61. Springer-Verlag, 1978.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM Press.

[BPIM04] Antonia Bertolino, Andrea Polini, Paola Inverardi, and Henry Muccini. Towards anti-model-based testing. 2004.

[Bri88] E. Brinskma. A theory for the derivation of tests. *Protocol Specification, Testing and Verification*, VIII:63–74, 1988.

[Bri06] Encyclopedia Britannica. Reactive systems (from computer), July 2006.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CES86a] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CES86b] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[Com82] Software Engineering Technical Committee. Glossary of software engineering terminology. IEEE Standard IEEE 729-1983, IEEE Computer Society, 345 East 47th Street, New York, NY 10017-2394, USA, September 1982.

[Com98] Software Engineering Technical Committee. Ieee standard for software test. IEEE Standard IEEE Std 829-1998, IEEE Computer Society, 345 East 47th Street, New York, NY 10017-2394, USA, September 1998.

[Com06a] Compuware. Compuware qadirector - advanced risk-based test management for distributed applications. http://www.compuware.com/products/qacenter/qadirector.htm, August 2006.

[Com06b] Compuware. File-aid/cs - complete test data management for distributed environments. http://www.compuware.com/products/fileaid/cs.htm, August 2006.

[Con06] Conformiq. Conformiq test generator. http://www.conformiq.com/ctg.php, July 2006.

[CS94] D. A. Carrington and P. A. Stocks. A tale of two paradigms: Formal methods and software testing. In J. P. Bowen and J. A. Hall, editors, *Workshops in Computing*, Z User Workshop, pages 51–68. Springer–Verlag, 1994.

[CS01] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. pages 1635–1790, 2001.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[dBRS+00] Lydie du Bousquet, Solofo Ramangalahy, Séverine Simon, César Viho, Axel Belinfante, and René G. de Vries. Formal test automation: The conference protocol with tgv/torx. In *TestCom*, pages 221–228, 2000.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[dFLC06] Gustavo da Fonseca Limaverde Cabral. Formal specification generation from requirement documents. Master's thesis, Centro de Informática - Universidade Federal de Pernambuco, 2006.

[Dil96] David L. Dill. The mur*hi* verification system. In *CAV*, pages 390–393, 1996.

[EFM97] André Engels, Loe M. G. Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 384–398, London, UK, 1997. Springer-Verlag.

[ETS96] ETSI. Methods for testing and specification (mts); test purpose style guide. ETSI TECHNICAL REPORT ETR 266, European Telecommunications Standards Institute, Office address: 650 Route des Lucioles - Sophia Antipolis - Valbonne - FRANCE, August 1996.

[FHNS02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 134–143, New York, NY, USA, 2002. ACM Press.

[FHP02] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, (41):89–110, 2002.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, London, UK, 1995. Springer-Verlag.

[Gau04] Marie-Claude Gaudel. Problems and methods for testing infinite state machines: Extended abstract. *Electr. Notes Theor. Comput. Sci.*, 95:53–62, 2004.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 112–122, New York, NY, USA, 2002. ACM Press.

[GHNS93] J. Grabowski, D. Hogrefe, R. Nahm, and A. Spichiger. Relating test purposes to formal specifications: Towards a theoretical foundation of practical testing, 1993.

[GM04] Glenford and J. Myers. *The Art of Software Testing*. John Wiley and Sons, New Jersey, 2004.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[GRS04] Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of asml, 2004.

[Har] Ian G. Harris. Hardware-software covalidation: Fault models and test generation.

[Hie01] R. Hierons. Checking states and transitions of a set of communicating finite state machines, 2001.

[HJ03] Holger Hermanns and Christophe Joubert. A set of performance and dependability analysis components for cadp. Tool Session of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland), April 2003.

[HKL97] Constance Heitmeyer, James Kirby, and Bruce Labaw. The scr method for formally specifying, verifying, and validating requirements: tool support. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 610–611, New York, NY, USA, 1997. ACM Press.

[HLSC01] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic test generation from statecharts using model checking, 2001.

[HN04] A. Hartman and K. Nagin. The agedis tools for model based testing. In *ISSTA*, pages 129–132, 2004.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol04] Gerard J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004. HOL g 03:1 1.Ex.

[How06] Rick Hower. Software qa and testing resources center - software qa and testing frequently-asked-questions, part 1. http://www.softwareqatest.com/qatfaq1.html, August 2006.

[HP98] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[HSSS96] Franz Huber, Bernhard Schatz, Alexander Schmidt, and Katharina Spies. Autofocus: A tool for distributed systems specification. In *FTRTFT*, pages 467–470, 1996.

[IBM06] IBM. Ibm software - rational robot - product overview. http://www-306.ibm.com/software/awdtools/tester/robot/, August 2006.

[Int06] IntelliJ. Intellij idea :: The most intelligent java ide. http://www.jetbrains.com/idea/, August 2006.

[Ire97] Barry L. Ires. Book review: Using z by jim woodcock and jim davies (prentice hall, 1996). *SIGSOFT Softw. Eng. Notes*, 22(4):114–115, 1997.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[Jak06] Jakarta. Jakarta cactus. http://jakarta.apache.org/cactus/, August 2006.

[JJ05] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[Jor95] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, second edition edition, 1995.

[JR96] Abrial Jean-Raymond. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[JUn06] JUnit.org. Junit, testing resources for extreme programming. http://www.junit.org/index.htm, August 2006.

[JW90] J.Baeten and W. Weijland. *Process Algebra*, volume 18 of *of Cambridge tracts in theoretical computer science*. Cambridge University Press, 1990.

[KL] Moatz Kamel and Stefan Leue. Validation of a remote object invocation and object migration in corba giop using promela/spin. In *International SPIN Workshop*.

[Kru00] Philippe Kruchten. *The Rational Unified Process An Introduction*. Addison-Wesley, second edition edition, 2000.

[Lam80] Leslie Lamport. "sometime"is sometimes "not never": on the temporal logic of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM Press.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[LdBB+01] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. *ase*, 00:127, 2001.

[Lei06] Daniel Almeida Leitão. Nlforspec: Uma ferramenta para geração de especificações formais a partir de casos de teste em linguagem natural. Master's thesis, Centro de Informática - Universidade Federal de Pernambuco, 2006.

[Lew00] William E. Lewis. *Software Testing and Continuous Quality Improvement*. Auerbach, 2000.

[Lin06] Linux.org. The linux home page at linux online. http://www.linux.org/, August 2006.

[LPB93] G. Luo, A. Petrenko, and G. V. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. Technical Report IRO-864, 1993.

[LvBP94] Gang Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Softw. Eng.*, 20(2):149–162, 1994.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[Mar95] Bruno Marre. Loft: A tool for assisting selection of test data sets from algebraic specifications. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 799–800, London, UK, 1995. Springer-Verlag.

[Mat03] Radu Mateescu. On-the-fly verification using cadp. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.

[Mer06a] Mercury. Performance monitor - mercury loadrunner performance monitor. http://www.mercury.com/us/products/performance-center/loadrunner/monitors/, August 2006.

[Mer06b] Mercury. Regression testing - mercury winrunner. http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/, August 2006.

[Mic06] Sun Microsystems. Java 2 platform standard edition 5.0. http://java.sun.com/j2se/1.5.0/, August 2006.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MIT06] MIT. Mulsaw project web page. http://mulsaw.lcs.mit.edu/, July 2006.

[MK01] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Nic87] R De Nicola. Extensional equivalence for transition systems. *Acta Inf.*, 24(2):211–237, 1987.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[oT06] Warsaw University of Technology. The object-oriented software testing environment (toster), July 2006.

[Par06] Parasoft. Jtest: Java unit testing. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, August 2006.

[Pet01] Alexandre Petrenko. Fault model-driven test derivation from finite state models: annotated bibliography. pages 196–205, 2001.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM Press.

[psI89] ISO 8807:1989 Information processing systems and Open Systems Interconnection. *LOTOS : A formal description technique based on the temporal ordering of observational behaviour*. ISO, 1989.

[PSSD00] David Y. W. Park, Ulrich Stern, Jens U. Sakkebæk, and David L. Dill. Java model checking. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 253, Washington, DC, USA, 2000. IEEE Computer Society.

[PY96] A. Parashkevov and J. Yantchev. Arc — a tool for efficient refinement and equivalence checking for csp, 1996.

[Que99] Geoff Quentin. Automated software testing: Introduction, management and performance, elfriede dustin, jeff rashka and john paul, addison-wesley, 1999 (book review). *Softw. Test., Verif. Reliab.*, 9(4):283–284, 1999.

[Rat06] Rational. Using rational pure coverage. http://bmrc.berkeley.edu/purify/docs/html/installing_and_gettingstarted/3-pureCov.html, August 2006.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[RW05] Holger Rasch and Heike Wehrheim. Checking the validity of scenarios in uml models. In *FMOODS*, pages 67–82, 2005.

[SAV+05] Augusto Sampaio, Carlos Albuquerque, Joåo Vasconcelos, Luckerson Cruz, Luis Figueiredo, and Sergio Cavalcante. Software test program: a software residency experience. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 611–612, New York, NY, USA, 2005. ACM Press.

[SC93] P. Stocks and D. Carrington. Test template framework: a specification-based testing case study. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 11–18, New York, NY, USA, 1993. ACM Press.

[Sch92] S. A. Schneider. An operational semantics for timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1991*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.

[Sch00] Steve Schneider. Abstraction and testing in csp. *Formal Asp. Comput.*, 12(3):165–181, 2000.

[SEG00] Michael Schmitt, Michael Ebner, and Jens Grabowski. Test Generation with Autolink and TestComposer. In *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000), Grenoble (France), June, 26 - 28, 2000*, June 2000.

[Soa06] Elifrancis Rodrigues Soares. Processo de análise de cobertura alinhado ao processo de desenvolvimento de software. Master's thesis, Centro de Informática - Universidade Federal de Pernambuco, 2006.

[Soc06] DL Forum Society. What is an msc? http://www.sdl-forum.org/MSC/index.htm, July 2006.

[Sof99] Rational Software. The rational approach to automated testing. page 15, 1999.

[Sof06] Quest Software. Java profiler for j2ee and java performance monitoring by quest software. http://www.quest.com/jprobe/, August 2006.

[Som01] Ian Sommerville. *Software engineering*. Number 6th. Addison-Wesley, 2001.

[SW92] Behcet Sarikaya and A. Wiles. Standard conformance test specification language ttcn. *Comput. Stand. Interfaces*, 14(2):117–144, 1992.

[Sys03] Formal Systems. *ProBE User Manual*. Formal Systems (Europe) Ltd, 2003.

[Sys05] Formal Systems. *Failures-Divergence Refinement - FDR2 User Manual*. Formal Systems (Europe) Ltd, June 2005.

[Tel06a] Telelogic. Compliance testing of communication systems using ttcn-2 - telelogic ttcn suite. http://www.telelogic.com/corp/products/tau/ttcn/overview.cfm, July 2006.

[Tel06b] Telelogic. Systems engineering, software development and testing automation - telelogic tau. http://www.telelogic.com/corp/products/tau/, July 2006.

[Tor06] Dante Gama Torres. Specnl: Uma ferramenta para gerar descrições em linguagem natural a partir de especificações de casos de teste. Master's thesis, Centro de Informtica - Universidade Federal de Pernambuco, 2006.

[Tre96a] J. Tretmans. Conformance testing with labelled transitions systems: Implementation relations and test generation. In *Computer Networks and ISDN*, volume 29, pages 49–79, 1996.

[Tre96b] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tre96c] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[Tre99] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[TVE06] TVEC. T-vec test vector generation system (tvgs). http://www.t-vec.com/solutions/tvec.php, July 2006.

[V.98] Hall P. A. V. Towards testing with respect to formal specification. In *In 2nd IEE/BCS Conference on Software Engineering*, pages 159–163, 1998.

[Ver06] CS Verilog. Objectgeode information - space tools cd. http://www.spacetools.com/tools4/space/213.htm, July 2006.

[Wik06a] Wikipedia. Finite state machine - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Finite_state_machine, August 2006.

[Wik06b] Wikipedia. Global system for mobile communications. http://en.wikipedia.org/wiki/Gsm, August 2006.

[Wik06c] Wikipedia. Short message service. http://en.wikipedia.org/wiki/Short_message_service, August 2006.

[Wik06d] Wikipedia. State transition system - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/State_transition_system, August 2006.

[WLPS00] Guido Wimmel, Heiko Loetzbeyer, Alexander Pretschner, and Oscar Slotosch. Specification based test sequence generation with propositional logic. *Software Testing, Verification & Reliability*, 10(4):229–248, 2000.

[WQ96] ISO/IEC JTC1/SC21 WG7 and ITU-T SG 10/Q.8. Information retrieval, transfer and management for osi; framework: Formal methods in conformance testing. ITU-T proposed recommendation Z.500 CD 13245-1, ISO - ITU-T, Geneve, 1996. Committee Draft.

[yWo06a] yWorks. Trivial graph format (tgf). http://www.yworks.com/products/yfiles/doc/developers-guide/tgf.html, August 2006.

[yWo06b] yWorks. yed : Java graph editor. http://www.yworks.com/en/products_yed_about.htm, August 2006.