Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Abordagem para Geração Automática de Código para Framework de Automação de Testes

Euclides Napoleão Arcoverde Neto

Dissertação de Mestrado

Recife
30 de Agosto de 2007

Universidade Federal de Pernambuco
Centro de Informática

Euclides Napoleão Arcoverde Neto

Abordagem para Geração Automática de Código para Framework de Automação de Testes

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 Ciência da Computação.

Orientador: *Prof. Dr. Alexandre Cabral Mota*

Recife
30 de Agosto de 2007

Agradecimentos

À minha noiva Emanuelle, pelo seu carinho, amor, incentivo e paciência que foram fundamentais para a conclusão desse trabalho.

Aos meus pais, Cleide e Franklin, pelo incentivo, ajuda incondicional, amor e por terem proporcionado todas as condições para que pudesse chegar até aqui.

À minha irmã e meu cunhado, Patrícia e Marcos, e aos meus sobrinhos Marcelo e Thiago que recentemente nasceram e deixaram a família mais feliz.

À minha "segunda família", Sr. Renato, D. Aluilda, Michellini, Luiz Claudio, Renata, Evandro e Maria Clara que me propuseram momentos felizes nos últimos cinco anos.

Ao professor Francisco Madeiro que me ensinou a pesquisar e, além de amigo, é um exemplo de pessoa.

Ao amigo Clélio Souza que esteve presente nos momentos mais difíceis durante o mestrado, além de suas valiosas sugestões que ajudaram na concepção desse trabalho.

A Cristiano Bertolini e Manoel Messias que me ajudaram na obtenção dos resultados apresentados nessa dissertação.

A Sidney Nogueira e Patrícia Muniz por sempre estarem disponíveis para tirar dúvidas sobre CSP.

À Ana Claudia Didier, gerente que procurou me ajudar como pôde para realização dessa pesquisa, aos colegas de trabalho do projeto BTC Test Automation e ao C.E.S.A.R pelo suporte financeiro.

À Motorola pela oportunidade e pela infra-estrutura oferecidas para obtenção de resultados.

Aos professores Paulo Borba e Augusto Sampaio cujos questionamentos e sugestões que foram fundamentais para o desenvolvimento desse trabalho.

Finalmente, agradeço especialmente ao professor Alexandre Mota por sua paciência, suas críticas e suas contribuições em todo o trabalho.

Resumo

Engenharia de software visa criar software de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais. Um de seus principais objetivos é obter um grau de qualidade mínimo que, em geral, significa uma baixa taxa de defeitos. Considerando que qualidade é crítico para o sucesso do software, o uso de testes vem crescendo. Testes objetivam revelar a presença de erros o mais cedo possível no ciclo de desenvolvimento. Embora teste de software seja uma atividade complexa, geralmente não é realizada sistematicamente devido a uma série de fatores como limitações de tempo, recursos e qualificação técnica dos envolvidos. Dessa forma, a automação de testes é uma tendência na área de testes.

Esse trabalho está inserido no contexto de um projeto de pesquisa realizado pela Motorola em parceria com o Centro de Informática da Universidade Federal de Pernambuco denominado *CInBTCRD* (*CIn and Brazil Test Center Research and Development Project*). Na Motorola, a automação de testes faz-se por meio de um *framework* denominado de TAF (*Test Automation Framework*), o qual simula a interação de um ser humano com um aparelho celular, além de poder capturar o estado e outras informações importantes do aparelho.

Testes reusam implementações já definidas no TAF para criar os scripts de teste, bem como novas implementações são criadas quando o reuso não é possível. Visto que bastante tempo é gasto para criar tais implementações, este trabalho propõe criar uma estratégia que, dado um script de teste, gere código para o TAF das funcionalidades ainda não implementadas automaticamente.

Para gerar tais códigos, usamos a linguagem formal CSP (*Communicating Sequential Processes*) como base. CSP foi criada para especificar e projetar comportamentos de sistemas concorrentes e distribuídos. CSP possui uma teoria de refinamentos associada, a qual é o alicerce de nossa proposta. O uso de refinamento em nossa proposta é relativamente simples: dado um teste em TAF descrito usando CSP e o comportamento de um celular também em CSP, a relação de refinamento só será satisfeita quando todos os elementos contidos no teste estiverem de acordo com o definido no comportamento do celular. Assim sendo, nesse trabalho usamos a ferramenta BxT (*Behavior Extractor Tool*) que desenvolvemos no contexto do projeto de pesquisa para extrair automaticamente um modelo CSP de um celular, bem como reusamos um outro trabalho que consegue representar casos de teste em CSP e criamos um algoritmo que usa esses elementos e a teoria da refinamento de CSP para completar certas partes do caso de teste com o auxílio do modelo do celular. Finalmente, o teste em CSP resultante é novamente escrito em TAF e dessa forma conseguimos atualizar o framework automaticamente.

Palavras-chave: Automação de Testes, Extração de Modelos Comportamentais, Geração de Código, CSP, Checagem de Refinamentos, Sistemas Embarcados

Abstract

Software engineering aims to create software in an economic way; it must be reliable and work efficiently in real machines. One of its main goals is to obtain a minimum quality level that means a low defects rate in general. Considering that quality is critical for the success of software, the use of tests is increasing. The main goal of software testing is to detect the presence of errors as soon as possible during the development lifecycle. Although software testing is a complex activity, in general it is not performed systematically because of a set of factors such as lack of time, resources and technical qualification of the staff. Then, test automation is a trend in the area of testing.

The context of this work is a research project between Motorola and the Informatics Center (CIn) of the Federal University of Pernambuco called *CInBTCRD* (*CIn and Brazil Test Center Research and Development Project*). Inside of Motorola, the test automation is performed by a framework called TAF (*Test Automation Framework*), which simulates the human interaction with a mobile phone. This framework is also able to capture the state and other important information about the device.

Test reuse implementations already defined in TAF to create test scripts, as well as new implementations must be created when the reuse is not possible. Since the developers spend a lot of time to create such implementations, this work intend to create a strategy that, given one test script, generates TAF code automatically considering the functionalities that were not implemented yet.

To generate such a code, we use the formal language CSP (Communicating Sequential Processes) as basis. CSP was designed to capture the behavior of concurrent and distributed systems. CSP has an associated refinement theory, which is the essence of our proposal. The use of refinement in our proposal is relatively simple: given a TAF test case described using CSP and the behavior of a mobile phone, also in CSP, the refinement relation only will be satisfied when all the elements contained in the test conforms to the behavior of the mobile phone. So, we present a tool we have created called BxT (*Behavior Extractor Tool*) that extracts the mobile phone behaviors in CSP automatically. Also, we show how to get TAF test scripts in CSP, using another work developed in Motorola research project context. We developed an algorithm that uses these two CSP elements and the CSP refinement theory in order to complete the test case description using the phone behavioral model as basis. Finally, the resulting test in CSP is written in TAF again and in this way we have managed to update the framework automatically.

Keywords: Test Automation, Behavioral Models Extraction, Code Generation, CSP, Refinement Verification, Embedded Systems

Sumário

1 Introdução **1**
1.1 Objetivos e contexto 5
1.1.1 O projeto *CInBTCRD* 6
1.1.1.1 Geração de Casos de Teste 7
1.1.2 Visão Geral da Abordagem 9
1.2 Organização da dissertação 11

2 CSP **13**
2.1 Definição de Processos 15
2.1.1 Processos Primitivos 15
2.1.2 Prefixo 16
2.1.3 Recursão 16
2.1.4 Escolha Externa e Interna 17
2.1.5 Escolha Condicional 18
2.1.6 Internalização de Eventos 18
2.1.7 Composição Sequencial 19
2.1.8 Interrupção 19
2.1.9 Paralelismo 20
2.2 Semântica Algébrica e Operacional 21
2.3 Semântica Denotacional e Refinamentos 22
2.4 Suporte de Ferramentas 23

3 Testes de Software **25**
3.1 Níveis 26
3.2 Metas para um Processo de Teste 27
3.3 Abordagens 28
3.4 Automação 30
3.4.1 TAF – *Test Automation Framework* 32

4 Geração de Código para um Framework de Automação de Testes **39**
4.1 Obtenção do Script de Teste Prototipado 42
4.1.1 Modelagem de Casos de Teste como Modelos Formais Abstratos 43
4.1.2 Modelagem de Casos de Testes Formais 44
4.1.2.1 Processamento de Linguagem Natural 44
4.1.2.2 Geração de Modelos Formais Abstratos 46
4.1.2.3 Associação de Eventos CSP com Comandos de Script 48

4.1.3 Tradução de Casos de Teste Abstratos em Scripts de Teste 48
4.2 Extração Automática do Modelo Comportamental 50
4.3 Obtenção de Representação CSP das UFs a Serem Desenvolvidas 53

5 Estudo de Caso **63**
5.1 Visão Geral da Abordagem 63
5.2 Aplicação do Método Proposto 69

6 Conclusão **77**
6.1 Trabalhos Relacionados 78
6.2 Trabalhos Futuros 81

Lista de Figuras

1.1 Fluxo de informações do projeto *CInBTCRD*. 7
1.2 Exemplo de script de teste TAF prototipado. 10

3.1 Modelo de desenvolvimento em V. 26
3.2 Telas. 33
3.3 Um fragmento de um caso de teste em TAF. 35
3.4 Diagrama de classes do TAF. 37
3.5 Exemplo de mapeamentos de UFs em arquivo XML. 37
3.6 Arquitetura de camadas do TAF. 37

4.1 Exemplo de captura de conteúdo gráfico e lógico do PTF para a tela do menu principal de um telefone celular. 41
4.2 Abordagem utilizada para modelagem de casos de teste. 45
4.3 Objetivo em alto nível *versus* sequência de ações. 46
4.4 Uma instância de um caso de teste automático (script de teste). 47
4.5 Exemplo de caso de teste abstrato (em notação CSP). 48
4.6 Mapeamento de um evento CSP em um comando de script. 49
4.7 Exemplo da tag *prototype*. 49
4.8 Engenho da ferramenta BxT. 51
4.9 Exemplo de telas de um telefone celular. 52
4.10 Especificação CSP obtida através da execução da BxT em um telefone celular cujo objetivo era iniciar uma conversação e em seguida sair da aplicação de IM. 53
4.11 Representação de um caso de teste em LTS. 54
4.12 Representação de uma implementação em LTS. 55
4.13 Exemplo de processos CSP (caso de teste abstrato e modelo comportamental). 56

5.1 Visão geral da abordagem proposta. 64
5.2 Fluxo do processo de geração de modelos formais a partir de documentos de requisitos. 64
5.3 Um exemplo de um *user view use case*. 65
5.4 Processo CSP gerado a partir do caso de uso apresentado na Figura 5.3. 66
5.5 Fragmento de um script de teste prototipado em TAF. 67
5.6 Exemplo de especificação CSP obtida através da execução da BxT. 68
5.7 Especificação CSP de um caso de teste prototipado. 70
5.8 Especificação CSP obtida após substituição do evento *prototype*. 73
5.9 Telas referentes à execução do caso de teste apresentado na Figura 5.7. 75

5.10 Conteúdo da implementação da UF LoginIMImp gerada automaticamente. 75

Lista de Tabelas

2.1 Processos primitivos e operadores algébricos de CSP. 15
2.2 Operadores de CSP_M correspondentes aos operadores primitivos de CSP. 24

3.1 Ferramentas de Automação de Teste. 31
3.2 Script de teste TAF *versus* PTF. 34
3.3 Exemplo de caso de teste escrito em inglês (LNC). 35

CAPÍTULO 1

Introdução

Engenharia de software é uma área da Ciência da Computação que objetiva a criação e utilização de sólidos princípios de engenharia a fim de obter software de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais [Pre04, Som01]. O próprio significado de engenharia já traz os conceitos de criação, construção, análise, desenvolvimento e manutenção. A *IEEE Computer Society* define engenharia de software como "(1) A aplicação de uma sistemática, disciplinada e quantificável abordagem para o desenvolvimento, operação, e manutenção de software, isto é, a aplicação de engenharia a software. (2) O estudo de abordagens como em (1)" [Ele90]. A engenharia de software [Pre04, Som01] surgiu com o intuito principal de resolver a *crise* na produção de software. O termo *crise de software* foi utilizado a partir da década de 70 quando a engenharia de software praticamente inexistia e foi utilizado para expressar as dificuldades do desenvolvimento de software frente ao rápido crescimento de demanda por software, da complexidade dos problemas a serem resolvidos e da inexistência de técnicas estabelecidas para o desenvolvimento de sistemas que funcionassem adequadamente ou pudessem ser validados. Um dos principais objetivos do desenvolvimento de software é a obtenção de um grau de qualidade mínimo que, na maioria dos casos, significa desenvolver software com baixa taxa de defeitos. Uma vez que esse objetivo é alcançado, os clientes ficam mais satisfeitos e os custos com manutenção são minimizados.

Considerando que a qualidade é um ponto fundamental para o sucesso do software, a disciplina de teste vem se tornando cada vez mais importante para a engenharia de software. Teste de software é uma das principais atividades realizadas para melhorar a qualidade de um produto em desenvolvimento. Seu principal objetivo é revelar a presença de erros o mais cedo possível no ciclo de desenvolvimento de software, buscando minimizar o custo da correção dos mesmos. O custo para correção de um erro na fase de manutenção é de sessenta a cem vezes maior do que corrigi-lo durante o desenvolvimento [Pre04]. Essa atividade tem apresentado progressivamente um maior grau de abrangência e de complexidade dentro do processo de desenvolvimento [Mye04].

Embora o teste de software seja uma atividade bastante complexa, geralmente ela não é realizada de forma sistemática devido a uma série de fatores como limitações de tempo, recursos e qualificação técnica dos envolvidos. Outros agravantes para a realização dessa atividade são a alta complexidade dos sistemas sendo atualmente desenvolvidos e a constante necessidade de sua rápida evolução. Teste pode ser visto como uma atividade dinâmica cujo principal 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 veredicto. Testar é uma tarefa muito laboriosa, podendo alcançar cerca de 50% do custo total do desenvolvimento do software [GM04]. Uma vez que os sistemas tendem a ser cada vez mais complexos, eles precisam

cada vez serem mais confiáveis, sendo necessário garantir a qualidade dos mesmos. A menos que sejam descobertas formas eficientes de testar os softwares, a percentagem nos custos de desenvolvimento destinadas aos testes irá crescer significativamente.

A automação de testes tem sido vista como a principal medida para melhorar a eficiência da atividade de teste, e várias soluções têm sido propostas para esta finalidade. A automação do teste consiste em repassar para o computador tarefas de teste de software que seriam realizadas manualmente. E isso se dá geralmente por meio do uso de ferramentas de automação de testes. Podem ser consideradas para a automação as atividades de geração e de execução de casos de teste [1][2] [Bin99, Kan97]. Quando executada corretamente, a automação de testes é uma das melhores formas de reduzir o tempo de teste no ciclo de vida do software, diminuindo o custo e aumentando a produtividade do desenvolvimento de software como um todo, além de, consequentemente, aumentar a qualidade do produto final. Estes resultados podem ser obtidos principalmente na execução do teste de regressão, que se caracteriza pelo teste de aplicativos já estáveis que passam por uma correção de erros, ou de aplicativos já existentes que são evoluídos para uma nova versão e suas funcionalidades são alteradas [Kan97].

Um processo de teste (como exemplo o fluxo de teste do RUP [Kru00]) consiste em um conjunto de atividades, papéis e artefatos para avaliar a qualidade do produto testado. Os principais problemas deste processo são alto custo, seleção de bons casos de teste e sua automação [BBC+03]. Existem várias técnicas comportamentais e estruturais que visam aumentar a manutenibilidade, configurabilidade e reusabilidade dos testes, gerar testes precisos de forma eficiente e efetiva, etc. Dentre essas técnicas, destaca-se a automação que visa tornar o processo de teste, de uma forma geral, mais ágil, menos suscetível a erros e menos dependente da interação humana. Em [Que99], mostra-se que a automação de várias atividades do processo de testes (desde o gerenciamento, passando pela geração até chegar a execução automática) provoca uma diminuição média de 75% do esforço total, quando comparado ao esforço das mesmas atividades realizadas manualmente. O presente trabalho visa atuar em uma área específica do processo de teste, mas com o objetivo maior de melhorar o processo como um todo.

Esse trabalho trabalho está inserido no contexto de um projeto de pesquisa realizado pela Motorola em parceria com o Centro de Informática da Universidade Federal de Pernambuco denominado *CInBTCRD* [SAV+05] (*CIn and Brazil Test Center Research and Development Project*) . O objetivo geral desse projeto é contribuir com todo o processo de testes da Motorola, automatizando a geração, seleção e avaliação de casos de teste para aplicações móveis. Já o presente trabalho tem por objetivo de propor uma abordagem que seja capaz de gerar código automaticamente para um framework para automação de testes desenvolvidos no BTC (*Brazil Test Center*), agilizando assim o processo de automação de novos casos de teste, de manutenção e *porting* dos casos de teste já automatizados.

Apesar de haver um consenso entre os especialistas dos ganhos que podem ser alcançados com o uso de uma boa estratégia de automação de testes, esta é uma área ainda pouco dominada

[1] Caso de teste é um conjunto de condições ou variáveis sob as quais um testador vai determinar se um requisito ou caso de uso de uma aplicação é parcial ou completamente satisfeita [GM04].

[2] Geração e execução de casos de testes podem ser vistos, respectivamente, como o processo automatizado de obtenção de casos de testes e a execução deles diretamente no sistema, sem que haja a necessidade de interação humana.

pela indústria de software [FdCD+04]. Deste modo, as empresas acabam atuando na automação de testes sem a definição de objetivos e expectativas claros e reais e sem a aplicação de técnicas apropriadas. Por consequência, têm-se constatado um grande número de fracassos nos esforços para a automação de testes [Bac99, Mar97, Hen98]. É necessário planejar bastante e verificar a viabilidade de automação[3] para cada caso de teste. Alguns pontos são extremamente importantes para a realização desse estudo. Casos de teste que são executados repetidamente (diariamente), casos de teste complexos, casos de testes que visam estressar a aplicação, ou seja, executar passos repetitivos por um longo período a fim de encontrar defeitos. Todos esses são exemplos de bons candidatos para automação. Porém, mesmo se enquadrando como um bom candidato a automação, alguns casos de teste exigem alta taxa de manutenção. Nesse caso, é necessário analisar com cuidado a viabilidade de geração de scripts para execução automática de testes.

A Motorola tem investido no desenvolvimento de um *framework* para automação de testes denominado TAF (*Test Automation Framework*) [KKR+07, Rec06]. O TAF é um *framework* para automação de testes orientado a objetos, projetado para suportar a automação de testes de integração e de sistema dos softwares embutidos em telefones celulares produzidos e desenvolvidos pela Motorola. O TAF pode ser visto como uma camada de abstração de outro *framework* de automação de testes, também usado na Motorola, denominado PTF (*Phone Test Framework*) [EV06]. As funções providas pelo PTF, entretanto, são de baixo nível de abstração; a utilização direta do PTF para automatizar um caso de teste produz *scripts* de teste ilegíveis e difíceis de serem mantidos e portados para serem executados em outros modelos de telefone. Apesar disso, o PTF representa uma base bastante apropriada para implementação de testes automáticos. A principal motivação para o desenvolvimento do TAF é a reutilização de código a partir da portabilidade de casos de teste. Nesse *framework* os scripts de teste que são executados em telefones de modelos diferentes, mas que implementam as mesmas funcionalidades, são exatamente os mesmos. Esses testes reutilizáveis são escritos no que denominamos *alto nível* de abstração, ou seja, sem considerar detalhes específicos de implementação, hardware, posição de teclas, etc. Para que seja possível manter os casos de teste automatizados imutáveis para diversos modelos de telefone e promover a reutilização de código de testes automatizados, o TAF disponibiliza uma API de funcionalidades de alto nível que por sua vez podem ter várias implementações de baixo nível para se adequar ao comportamento de cada produto a ser testado.

Apesar de ser um *framework* bastante maduro, o TAF não é um produto fechado. Quando a automação de testes com o TAF foi iniciada, o maior desafio era criar uma arquitetura que permitisse o reuso e a extensibilidade (os princípios de arquitetura do TAF serão apresentados no Capítulo 3). Uma vez que esses princípios foram implementados, o TAF pôde então ser visto como uma camada de abstração do PTF. Nós observamos que o código das funcionalidades do TAF (denominadas *Utility Functions*, ou simplesmente UFs) é composto de chamadas de métodos do PTF e de algumas chamadas de métodos do próprio TAF. Basicamente, para cada tela, o PTF provê um conjunto de nomes, posição, estado, etc. de cada item da tela. Depois de

[3] A viabilidade de automação é verificada através de uma análise nas variáveis envolvidas no processo de automação, como por exemplo, o número de ciclos que irão executar aquele caso de teste, se o caso de teste leva muito tempo para ser executado, etc. Nessa análise, deve ser considerado o tempo gasto para automatizar cada caso de teste e se os ganhos introduzidos pela automação valerão a pena.

analisar o código de várias UFs, nós verificamos alguns padrões de código que são utilizados em praticamente todas as UFs. A partir daí, iniciamos uma análise mais profunda com o objetivo de verificar a viabilidade da criação de novas UFs automaticamente. O resultado dessa análise gerou a idéia do trabalho aqui apresentado.

Atualmente o desenvolvimento de novas UFs para o TAF é feito sob demanda. É importante salientar que praticamente todo o tempo gasto para desenvolver novos casos de teste ou portar casos de teste para novos produtos no TAF está relacionado à criação de novas funcionalidades. Para criar novos casos de teste no TAF, os desenvolvedores iniciam a composição de um caso de teste através da utilização de funcionalidades já existentes no *framework*. Caso seja necessário criar uma nova funcionalidade, o desenvolvedor utiliza um método especial chamado *prototype* que, durante a execução, pausa o fluxo do teste até que o desenvolvedor o libere. Dessa forma o caso de teste pode ser completamente validado, pois no momento que um método *prototype* é executado, o TAF pausa sua execução e avisa ao desenvolvedor que aquele passo deve ser executado manualmente a fim de que o caso de teste em desenvolvimento seja completamente validado (através da execução manual dos passos que representam funcionalidades ainda não implementadas no TAF). Uma vez que o caso de teste está totalmente validado, a próxima etapa do processo é justamente o desenvolvimento dessa nova funcionalidade no *framework*. Para portar casos de teste já existentes para novos produtos, o processo é um pouco diferente. O desenvolvedor inicia o *porting* executando o caso de teste a fim de encontrar algum ponto de alteração nas implementações atualmente existentes para as funcionalidades do TAF. Caso nenhuma alteração seja necessária, o caso de teste é dito portado e já está pronto para ser executado. No entanto, caso alguma funcionalidade não se adeque ao novo produto, o mesmo processo para criação de uma nova funcionalidade é executado, ou seja, adiciona-se um método *prototype* no caso de teste para cada ponto de falha. Ao final, todas as funcionalidades que precisam ser ajustadas são implementadas. A vantagem do *porting* em relação à criação de novos caso de teste diz respeito à diferença de tempo gasto na criação de uma nova funcionalidade desde o início (no caso de novas funcionalidades do TAF) e do tempo gasto para estender as funcionalidades já existentes a fim de resolver somente as diferenças (que é bem menor).

Aplicações que rodam em dispositivos móveis podem conter funcionalidades complexas, incluindo comportamento concorrente e troca de mensagens. Especificar o comportamento de sistemas com essa natureza pode ser uma tarefa complexa, especialmente se for necessário validar esse comportamento. Analisar elementos concorrentes, rodando em paralelo, e validar seus comportamentos pode ser uma atividade bastante laboriosa. Nesse caso, a utilização de uma linguagem de especificação formal para especificar esses comportamentos e verificá-los através de ferramentas é bastante interessante. No presente trabalho utilizamos a linguagem de especificação formal denominada CSP [Hoa85, RHB97] para especificar os modelos comportamentais dos telefones celulares, bem como os casos de teste em desenvolvimento.

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; conta com modelos semânticos capazes de representar o comportamento de ângulos variados, pelos quais ferramentas podem verificar propriedades das especificações, tais como: ausência de *deadlock*, ausência de *livelock*, comportamento

determinístico e refinamentos entre processos.

Considerando que CSP é o formalismo padrão adotado no projeto de pesquisa (apresentado na Seção 1.1.1), surgiu a oportunidade de se investigar o uso de CSP para geração automática de código para UFs do TAF, considerando o reuso de outros trabalhos existentes no *CInBTCRD*. Um desejo antigo dos gerentes das equipes de automação de testes da Motorola era justamente o de desenvolver UFs para o TAF automaticamente. Esse trabalho apresenta uma abordagem com essa finalidade.

A seguir, na Seção 1.1, serão apresentados os objetivos do presente trabalho, bem como o contexto no qual o mesmo está relacionado. É importante salientar que atualmente existem trabalhos para gerar scripts de teste para o TAF automaticamente [dS07], porém nenhum trabalho até então havia se proposto a criar também implementações das funcionalidades não existentes no TAF. O nosso grupo de pesquisa (*CInBTCRD*) está trabalhando para melhorar o processo de testes como um todo. Atualmente existem dois trabalhos que são fundamentais para o trabalho descrito nessa dissertação: o primeiro visa gerar automaticamente casos de teste escritos em Linguagem Natural Controlada (LNC), que é um subconjunto de uma linguagem tradicional, como Inglês ou Português, baseado nos documentos de requisitos; o segundo trabalho tem por objetivo a geração de casos de teste para o TAF também de forma automática assumindo reuso de funcionalidades TAF já existentes, bem como usadas em scripts de teste. Considerando esses dois trabalhos, um terceiro seria de fundamental importância para a melhoria do processos de testes como um todo: a geração de código de novas funcionalidades a fim de minimizar o tempo gasto na automação de testes usando o TAF. O presente trabalho propõe uma abordagem para o desenvolvimento dessas funcionalidades automaticamente, assumindo que reuso de funcionalidades TAF não foi possível durante o processo de geração de scripts de teste apresentado em [dS07]. Essa geração de novas funcionalidades é baseada em modelos CSP que representam o comportamento dos telefones que também são extraídos de forma automática.

1.1 Objetivos e contexto

Visto que o processo de automação de testes é uma tarefa não trivial e requer bastante tempo para criação de um ambiente estável (confiável) para execução de testes automáticos, o presente trabalho propõe a criação de uma ferramenta que, dado um caso de teste semi-automatizado[4], gere automaticamente código para as funcionalidades ainda não implementadas no TAF[5].

O projeto *CInBTCRD* surgiu da necessidade de se complementar os outros projetos do BTC. Durante o dia-a-dia, várias equipes de desenvolvimento do BTC identificavam oportunidades de melhoria no processo de desenvolvimento e execução de testes. Porém, devido à correria do trabalho e aos cronogramas apertados, essas equipes normalmente não têm tempo para explorar as possíveis soluções. O *CInBTCRD* aparece nesse contexto para explorar em detalhes essas oportunidades, propondo e implementando soluções para o processo de testes do BTC. A

[4]Utilizamos o termo semi-automatizado para denotar scripts de teste prototipados. Especificamente para esse trabalho estamos falando de um script de teste TAF que contém pelo menos um elemento *prototype* em seu código.

[5]Apesar de estarmos utilizando o TAF como base de desenvolvimento, futuramente pretendemos estender essa proposta para outros *frameworks* de automação de testes. Admitimos que os leitores dessa dissertação tenham conhecimento em Java [Gra97] e UML (*Unified Modeling Language*) [Mar03].

seguir, o projeto será contextualizado mais detalhadamente.

1.1.1 O projeto *CInBTCRD*

Procuramos dar uma visão geral de todos os trabalhos atualmente sendo realizados dentro desse projeto, e também destacamos onde o presente trabalho está inserido. O *CInBTCRD* atualmente tem seguintes objetivos específicos:

- Documentação de Requisitos – desenvolvimento de uma Linguagem Natural Controlada (LNC) a ser utilizada na documentação de requisitos, com o intuito de sistematizar a geração e seleção efetiva de casos de teste.

- Seleção de Casos de Teste – definir um procedimento bem definido para seleção de casos de teste, tornando possível a identificação efetiva de testes com potencial para revelar erros importantes na aplicação, e com cobertura adequada.

- Requisitos Documentados a partir dos Testes – muitas vezes, os casos de teste contêm informações mais atualizadas do que a documentação de requisitos. A geração/atualização de requisitos a partir dos casos de testes é um outro importante objetivo desta iniciativa.

- Avaliação da Suíte de Testes e Resultados – o escopo do projeto inclui ainda o desenvolvimento de técnicas e ferramentas que permitam analisar parâmetros como cobertura e confiabilidade dos testes e estimar o tempo de execução dos casos de teste gerados.

A Figura 1.1 exibe o fluxo de informação do projeto *CInBTCRD*. Os quadros representam os artefatos gerados pelas atividades que estão sendo desenvolvidas no projeto. As atividades são representadas pelas setas numeradas.

As atividades do *CInBTCRD* podem ser divididas em três grandes grupos, como a seguir:

- **Geração de Casos de Teste:** agrupa as atividades representadas pelas setas 1, 2, 5 e 9, e tem como objetivo gerar casos de teste a partir de documentos de requisitos e de diagramas UML. As atividades e artefatos que compõem este grupo serão detalhados na Seção 1.1.1.1. Essas atividades fazem parte do processo de engenharia direta realizada pelo projeto.

- **Atualização dos Requisitos:** agrupa as atividades representadas pelas setas 3, 4 e 6, e tem como objetivo atualizar os requisitos (documentos e diagramas) a partir de informações mais atuais contidas nos casos de teste. As atividades e os artefatos que compõem este grupo podem ser encontrados nos trabalhos [Lei06, dS07]. Esse grupo de atividades compõe o processo de engenharia reversa realizada pelo *CInBTCRD*.

- **Estimativas de Execução e Cobertura de Código:** agrupa as atividades 7 e 8. Recebe como entrada casos de teste e tem como objetivo estimar a execução desses testes e analisar a sua cobertura de código. Maiores detalhes dessas duas atividades serão podem ser encontrados em [AB07b, AB07a, ABL06, Soa06].

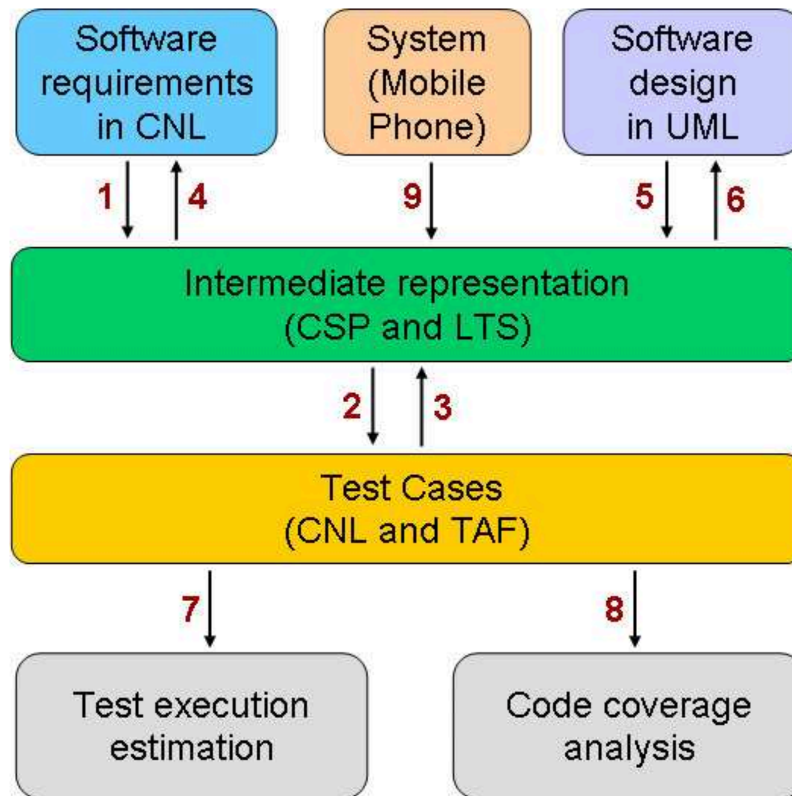


Figura 1.1 Fluxo de informações do projeto *CInBTCRD*.

A seguir, serão brevemente descritas as atividades relacionadas ao primeiro grupo que estão relacionadas ao presente trabalho. A numeração indicada pelas setas da Figura 1.1 continuará sendo utilizada a seguir para facilitar o entendimento do leitor.

1.1.1.1 Geração de Casos de Teste

A atividade de Geração de Casos de Teste dentro do *CInBTCRD* utiliza a abordagem conhecida como Teste Baseado em Modelos (*Model-Based Testing – MBT*) [EFW01]. MBT é uma técnica para geração de teste de software que consiste em especificar um modelo formal ou semi-formal dos requisitos da aplicação a ser testada de modo a caracterizar com exatidão o seu comportamento e, a partir desse modelo especificado, gerar os testes [DJK+99].

No *CInBTCRD*, a estratégia para geração de testes é baseada na abordagem MBT. Os modelos formais utilizados no processo de geração são produzidos a partir da documentação do sistema em teste. Na Figura 1.1, as atividades 1, 5 e 9 geram uma representação intermediária (modelo formal) a partir dos requisitos escritos em linguagem natural controlada, de diagramas do sistema e do próprio sistema sob teste. A partir daí, a atividade 2 gera os casos de testes com base nessa representação intermediária. A seguir, essas três atividades que compõem a estratégia do projeto serão brevemente apresentadas.

Geração de Modelos de Uso a partir de Documentos de requisitos (atividade 1) – Esta atividade é considerada a primeira no processo de geração de testes. Aqui, as funcionalidades do sistema são especificadas em casos de uso escritos em uma linguagem natural controlada (LNC). Essa LNC é um subconjunto do Inglês (linguagem natural) que pode ser processado diretamente por uma máquina, além de ser expressiva o suficiente para ser entendida por não especialistas. Para padronizar a documentação de casos de uso, um *template* no formato Microsoft Word foi definido. A mesma LNC usada na escrita casos de uso também será usada na edição dos casos de teste.

Com as funcionalidades do sistema (casos de uso) documentadas na LNC, é possível realizar o processamento automático da documentação com a finalidade de gerar o modelo de uso do sistema. Este modelo de uso é uma representação formal das possíveis ações do usuário sobre o sistema. O modelo é escrito na notação formal CSP [Hoa85, RHB97] (Capítulo 2), sendo representado pela *Intermediate Representation* na Figura 1.1. Mais informações sobre essa atividade podem ser encontradas em [dFLC06, CS06].

Geração de Casos de Teste (atividade 2) – Essa tarefa representa o propósito geral do projeto de pesquisa: a geração automática de teste. Essa atividade recebe como entrada o modelo de uso em CSP do sistema em teste e gera automaticamente os casos de teste, também especificados em CSP. A geração é guiada por propósitos de teste, que direcionam a geração de acordo com critérios de cobertura de requisitos [dCN06]. Os benefícios diretos dessa geração automática é o aumento da eficiência e da produtividade, bem como da qualidade dos testes gerados.

Porém, os casos de testes gerados ainda estão sendo representados na *Intermediate Representation* (descritos na notação formal CSP). Com o objetivo de tornar a notação formal CSP transparente para os engenheiros de teste, os casos de teste em CSP são mapeados automaticamente para a LNC [Tor06]. Isso aumenta a eficiência da execução manual dos testes gerados, visto que os engenheiros não perdem tempo interpretando a especificação formal.

Além de gerar casos de testes em LNC, também são gerados *scripts* de teste para o TAF [dS07]. A tradução de casos de teste abstratos (em CSP) em scripts de teste para o TAF usa uma tabela de mapeamentos entre os eventos correspondentes a casa passo do teste com o código TAF para representar aquele evento. A idéia é obter os casos de teste em CSP e extrair todos os eventos correspondentes ao caso de teste TAF. Um parser foi desenvolvido para capturar esses eventos e enviá-los para o mecanismo de busca. Após extrair todos os comandos necessários para compor o caso de teste TAF, os mesmos são organizados em arquivos separados. O resultado é a representação do caso de teste automatizado em TAF. Essa representação do caso de teste TAF tem uma representação correspondente em CSP que é uma das entrada para o trabalho descrito nessa dissertação.

Extração de Comportamento dos Telefones Celulares (atividade 9) – Essa atividade tem por objetivo extrair um modelo formal através do próprio sistema. O principal objetivo dessa abordagem é a geração de um modelo formal através da própria implementação sob teste, sendo assim possível fazer checagem de modelos que representam os requisitos e a implementação em substituição à execução de casos de testes [BM07]. Um mecanismo para escanear

todos os caminhos do sistema de um telefone celular Motorola e extrair todo o comportamento do sistema foi desenvolvido nessa atividade. A saída dessa atividade é outra entrada para o trabalho proposto nessa dissertação.

1.1.2 Visão Geral da Abordagem

Depois da apresentação geral dos trabalhos relacionados no *CInBTCRD*, apresentaremos agora uma discussão mais detalhada sobre a metodologia proposta nessa dissertação.

A execução de testes automáticos ajuda a reduzir esforço, verificar padrões e melhorar a qualidade do produto sob teste. Por outro lado, a automação de testes é cara e precisa ser cuidadosamente estimada para que não traga prejuízo às organizações. Considerando o *framework* de automação de testes utilizado na Motorola, o TAF, praticamente todo o tempo gasto para desenvolver novos casos de testes está relacionado à criação de novas funcionalidades para o *framework*. Só após o finalizado o desenvolvimento dessas funcionalidades é que a camada de casos de teste tem acesso às mesmas.

Até então não existia nenhum trabalho dentro do grupo de pesquisa do BTC com o objetivo de desenvolver *Utility Functions* para o TAF automaticamente. O grupo de pesquisas vem trabalhando para melhorar o processo de teste de software como um todo. Uma oportunidade de contribuição nesse aspecto seria justamente a geração automática do código correspondente às novas funcionalidades a serem introduzidas no TAF. Dois trabalhos existentes no grupo de pesquisa nos serviram de motivação. O primeiro objetiva a geração de casos de teste escritos em linguagem natural controlada (LNC), baseado em documentos de requisitos também escritos em LNC [CS06, dCN06, Car06, Lei06]. Essa geração é baseada no processamento de uma representação intermediária (modelo CSP) dos requisitos. A saída desse processamento é um conjunto de testes representados em LNC (e cada caso de teste tem uma representação intermediária em CSP). O segundo trabalho tem por objetivo a geração de casos de testes automatizados para o TAF para cada um dos casos de teste gerados pelo primeiro trabalho [dS07]. Nesse trabalho o autor usa a representação intermediária de cada caso de teste a fim de atualizar os requisitos e obter scripts de teste automatizados em TAF. A exemplo do primeiro trabalho, uma representação CSP de cada caso de teste gerado para o TAF também é mantida. É importante mencionar que último trabalho visa gerar somente o código para a camada de casos de teste do TAF. Isto é, para cada funcionalidade não encontrada na base de dados do TAF, o autor representa essa funcionalidade com um evento especial denominado *prototype*, como pode ser visto no *Step 2.P Login* da Figura 1.2. Porém não fazia sentido escrever os requisitos em LNC, derivar casos de teste, codificar esses casos de teste em scripts TAF e não poder executá-los caso alguma funcionalidade necessária não estivesse presente no TAF. O trabalho aqui proposto tem por objetivo preencher essa lacuna entre a geração automática de casos de teste para o TAF e sua execução.

Nessa dissertação é apresentada uma abordagem para desenvolver automaticamente essas UFs baseado no refinamento entre modelos CSP que representam o comportamento dos telefones celulares e os modelos correspondentes aos scripts de teste prototipados. Para obtenção do modelo comportamental, nós desenvolvemos a ferramenta BxT. O presente trabalho considera que a implementação atual do dispositivo móvel está correta, sendo assim, admitimos que a BxT sempre gera um modelo formal correto do software embarcado. Nesse momento o leitor

```
// Step 1.P: Start IM application
navigationTk.launchApp(PhoneApplication.IM);

// Step 1.ER: IM application is started
phoneTk.checkScreen(PhoneScreen.IM);

// Step 2.P: Login
prototype("imTk.loginIM()");

// Step 2.ER: Contact List is shown
phoneTk.checkScreen(IMScreen.CONTACT_LIST);

// Step 3.P: Select a contact
imTk.scrollToContact(IMContact.JOHN);

// Step 3.ER: Conversation is opened
phoneTk.checkScreen(IMScreen.CONVERSATION);

// Step 4.P: Exit IM application
imTk.exitIM();

// Step 4.ER: Phone goes to idle
phoneTk.checkScreen(PhoneScreen.IDLE);
```

Figura 1.2 Exemplo de script de teste TAF prototipado.

pode estar se perguntando qual a contribuição em gerar a implementação de uma UF para executar um script de teste considerando que o software não tem erros. Porém, a nossa proposta é persistir a implementação da UF criada no TAF. Ou seja, uma vez que uma implementação de UF é gerada, a mesma é executada diversas vezes a cada ciclo de teste para todos os produtos da mesma família e essa nova implementação acusará falhas caso o comportamento dos produtos da mesma família mude.

O evento *prototype* presente nos scripts de teste gerados automaticamente é fundamental para a obtenção dos resultados apresentados no Capítulo 5. Esse evento representa as UFs do TAF que precisam ser implementadas para que o caso de teste (até então) parcialmente implementado possa ser executado. Para tanto, precisamos encontrar a implementação correta para cada *prototype* os scripts de teste e adicionar essa implementação na base de UFs do TAF. Assim, criamos um algoritmo que utiliza o resultado do refinamento entre os modelos CSP (modelo comportamental e do script de teste), a fim de encontrar o código correspondente a cada *prototype*. O nosso objetivo é gerar uma implementação que satisfaça cada *prototype* e atualize o script de teste automaticamente, substituindo todos os eventos *prototype* pelas suas respectivas UFs, e assim fazendo com que ele possa ser executado sem que seja necessária a intervenção humana para a implementação dessas funcionalidades ainda não presentes no TAF.

Porém o algoritmo proposto não é capaz de resolver todos os casos. A seguir descreveremos alguns cenários que não são cobertos pela abordagem proposta nessa dissertação:

- Não faz sentido para o algoritmo existirem dois eventos *prototype* em sequência, pois

não é possível saber onde um começa e o outro termina.

- Também não é possível identificar a possível implementação de um *prototype* caso este seja o último evento de um script de teste, pois não necessariamente a BxT produzirá um modelo comportamental que termine no mesmo ponto (modelos tendem a ser recursivos). Nesse caso, o algoritmo muito provavelmente não produzirá uma saída válida.

- O algoritmo proposto também não considera a possibilidade de obter múltiplas implementações para um determinado *prototype*. Esse caso pode ocorrer quando o modelo disponibiliza diversos caminhos diferentes para chegar no mesmo ponto (que seria o objetivo da UF). Nesse caso, o algoritmo que desenvolvemos considera somente o primeiro contra-exemplo retornado pela relação de refinamento executada pela ferramenta que utilizamos para tal, ou seja, o algoritmo acha a menor implementação para a UF representada pelo *prototype* em questão.

A seguir apresentaremos as principais contribuições do presente trabalho e, em seguida, na Seção 1.2, apresentaremos como está organizada essa dissertação.

- Criamos um algoritmo capaz de gerar a implementação de uma funcionalidade não existente em um *framework* de automação de testes. Esse algoritmo utiliza a teoria de refinamento de modelos CSP e fornece uma sequência de passos que, quando executada, representa a sequência necessária para exercitar o caso de teste já automatizado.

- Elaboramos uma estratégia que possibilita a execução automática de um caso de teste obtido diretamente através do processamento dos documentos de requisitos.

- Aplicamos um estudo de caso para demonstrar o funcionamento do algoritmo que correspondeu com as nossas expectativas de, inicialmente, propor uma abordagem que fosse capaz de substituir o esforço humano no desenvolvimento de novas funcionalidades relativamente simples para o TAF.

1.2 Organização da dissertação

Este capítulo é a introdução da dissertação e apresenta o seu contexto, incluindo problemas, principais objetivos e uma visão geral da solução proposta. A seguir o conteúdo de cada um dos capítulos será brevemente descrito.

O Capítulo 2 apresenta o formalismo CSP, incluindo os diversos operadores utilizados para definir processos, uma visão sobre a semântica denotacional e operacional e suas relações através de exemplos. Ao final do capítulo, é mostrada uma visão geral sobre os três modelos semânticos de CSP que são usados para estabelecer as relações de refinamento entre processos. As relações de refinamento são de fundamental importância para a obtenção de resultados do presente trabalho.

O Capítulo 3 fornece uma introdução sobre teste de software apresentando as definições básicas e a relação de Testes com Garantia da Qualidade de Software. Caracteriza as diversas fases de teste, as técnicas, as abordagens e as estratégias mais comuns. No fim do capítulo,

uma visão geral sobre os cuidados que devem ser tomados e os benefícios introduzidos ao se automatizar casos de teste, bem como uma visão geral do TAF (*Test Automation Framework*), que é utilizado na Motorola para automação de testes de integração e de sistema.

O Capítulo 4 mostra a principal contribuição do presente trabalho. Apresentamos como obtemos a representação do modelo de cada caso de teste a ser desenvolvido e em seguida apresentamos uma ferramenta denominada BxT (*Behavior Extractor Tool*), que tem por objetivo extrair um modelo formal, em CSP, a partir das implementações dos telefones celulares da Motorola. Mostramos que esses dois modelos obtidos servem de entrada para o algoritmo proposto que tem como base a verificação de refinamento de CSP. Finalmente é mostrado um exemplo de utilização do algoritmo a fim de gerar a implementação correspondente à UF do TAF para o script de teste prototipado em questão.

O Capítulo 5 mostra um estudo de caso realizado para que o método proposto nessa dissertação fosse validado.

Finalmente, no Capítulo 6 são apresentados as considerações finais, os trabalhos relacionados e os trabalhos futuros.

CAPÍTULO 2

CSP

CSP [Hoa85, RHB97] (*Communicating Sequential Processes*) é uma notação formal utilizada para descrever sistemas concorrentes e distribuídos onde os componentes interagem entre si através de um mecanismo de comunicação. CSP pertence a uma classe de formalismo chamada **álgebras de processo** cuja formulação reside sobre um rico conjunto de leis algébricas e cálculos para verificação de propriedades de especificações. Uma forma de entender CSP é imaginar um sistema como uma composição de unidades comportamentais independentes (subsistemas, componentes ou simplesmente rotinas de processamento) que comunicam-se entre si e com o ambiente que os cerca através de canais [Hoa85]. Cada uma destas unidades independentes pode ser formada por unidades menores, combinadas por algum padrão de interação específico usando os operadores de processo de CSP. A versão de CSP que nos baseamos é a de Roscoe [RHB97], que introduz algumas modificações sobre a versão original de Hoare [Hoa85]. CSP tem uma versão para uso em ferramentas denominada de CSP_M (*Machine Readable CSP*) [Sca98], a qual será usada nos capítulos posteriores desse trabalho. Neste capítulo apresentamos a versão original de CSP e no final apresentamos uma tabela mostrando como CSP_M representa o CSP que usamos aqui.

Processos são abstrações para unidades comportamentais e são construídos através de eventos, operadores e outros processos. Isto é, processos podem ser combinados para formar processos maiores, até que o comportamento de todo o sistema esteja especificado. A composicionalidade de processos (facilidade para compor processos complexos a partir de processos menores, sem alterar a estrutura interna das partes componentes) permite que CSP seja satisfatoriamente usada para modelar sistemas bastante complexos.

Eventos são abstrações de ações do mundo real. Um evento 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 no mundo real, um evento representado em CSP ocorre instantaneamente. Em CSP é dito que um evento simplesmente ocorreu, ou seja, 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. Restrições de tempo sobre a ocorrência dos eventos não são levados em conta (*untimed events*). Considera-se apenas a ordem em que os eventos ocorrem. Existem extensões de CSP em que o tempo é considerado (*Timed CSP*) [Sch92], porém estas não são consideradas no presente trabalho.

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

$\texttt{channel}\ button : \texttt{Int}$

Através desta, conseguimos introduzir o canal *button* que permite 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. Ou seja, o canal *button* pode ser visto como o conjunto de eventos

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

onde n é o valor máximo para o tipo `Int` predefinido. Considere o canal c, o operador de expansão aplicado ao canal c, denotado por $|\ c\ |$[1], retorna um conjunto com a enumeração de todos os eventos formados com base na declaração desse canal. 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:

$$MAX_OPTIONS\ =\ 4$$
$$OPTIONS\ =\ \{1..MAX_OPTIONS\}$$
$$\texttt{channel}\ option\ :\ OPTIONS$$

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 dessa constante é 4, o conjunto formado pela expansão do canal *option* é igual a $\{option.1,\ option.2,\ option.3,\ option.4\}$.

Esses eventos especificam o pressionamento de um seletor com quatro opções, em que $option.i$, $1 \leq i \leq 4$, corresponde à seleção da opção i.

A declaração de um canal sem tipo, como

$$\texttt{channel}\ turnoff$$

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

Todos esses eventos fazem parte de um **alfabeto**[2], denotado pela letra α, 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 α_P (α acrescido do nome do processo subscrito). O valor do alfabeto α_P 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 estiver sincronizando com outro(s) processo(s), o processo P estará obrigatoriamente

[1] Vale salientar que só faz sentido usar $|\ c\ |$ dentro de um conjunto, ou seja, $\{|\ c\ |\}$.

[2] Hoare [Hoa85] representa o alfabeto de cada processo com a letra α, enquanto Roscoe [RHB97] utiliza a letra Σ para representar o alfabeto da especificação inteira.

Sintaxe do Processo	Descrição
Stop	(quebra ou *deadlock*)
Skip	(término com sucesso)
$a \rightarrow P$	(prefixo)
$P(s)$	(recursão)
$P(s) \ \Box \ P(s)$	(escolha externa)
$P(s) \sqcap P(s)$	(escolha interna)
$\texttt{if } (g) \texttt{ then } P(s) \texttt{ else } P(s)$	(escolha condicional)
$P(s) \setminus C$	(internalização)
$P(s) \ ; \ P(s)$	(composição seqüencial)
$P(s) \bigtriangleup P(s)$	(interrupção)
$P(s) \vert[C]\vert P(s)$	(paralelismo)

Tabela 2.1 Processos primitivos e operadores algébricos de CSP.

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 é descrita através dos operadores algébricos de CSP. A sintaxe de CSP define a forma como eventos e processos podem ser combinados através de operadores para formar novos processos. A seguir, apresentamos a sintaxe considerada neste trabalho [RHB97] seguida de uma breve descrição para cada um dos elementos sintáticos a serem utilizados.

2.1 Definição de Processos

Cada processo é definido através de 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 esquerdo introduz o nome e um ou mais parâmetros, e o lado direito o corpo do processo. Como exemplo, considere que o lado esquerdo da equação de um processo parametrizado P é representado como $P(s)$, então s é o parâmetro de P. Os parâmetros de um processo, juntamente com os parâmetros de entrada de eventos, representam o estado interno do processo.

A Tabela 2.1 contém uma listagem dos operadores algébricos e os processos primitivos de CSP que são utilizados para compor equações de processos mais elaborados. Nesta Tabela, considere g uma expressão condicional (que retorna um booleano), a um evento ($a \in \Sigma$) e C um conjunto de eventos. A seguir descreveremos brevemente cada um dos elementos da Tabela 2.1.

2.1.1 Processos Primitivos

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

execução com sucesso e sua utilidade aparecerá mais claramente na descrição do operador de composição sequencial (;) na Seção 2.1.7.

2.1.2 Prefixo

Para construir um processo através de seus eventos usamos o operador $\rightarrow$ (chamado operador de prefixo). O operador de prefixo é utilizado com um evento do lado esquerdo e um processo do lado direito (ver Tabela 2.1). Como exemplo, temos o comportamento do processo VM_1, que é definido por um prefixo

$$VM_1 = coffee \rightarrow VM_2$$

esse processo oferece o evento *coffee* ao ambiente e aguarda indefinidamente até que o ambiente esteja pronto para sincronizar neste evento. Quando isso acontece, o processo passa a comportar-se como o processo VM_2. Também é possível criar um processo com vários prefixos em seqüência, como no exemplo do processo VM_2:

$$VM_2 = coin \rightarrow coffee \rightarrow VM_3$$

O processo VM_2 aguarda que o ambiente esteja pronto para sincronizar no evento *coin*, após a sincronização e comunicação desse evento se comporta como o processo $coffee \rightarrow VM_3$, este último aguarda a sincronização no evento *coffee* para em seguida se comportar como o processo VM_3. O aninhamento de prefixos (um prefixo definido por outro) pode ser empregado ilimitadamente.

2.1.3 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 si próprio, como o processo $P = a \rightarrow P$. Este processo oferece o evento a e espera que o ambiente sincronize, após a sincronização se comporta de acordo com a recursão P (como si próprio indefinidamente). É também útil para definir processos que possuem recursão mútua entre si, como o exemplo a seguir.

$$VM_3 = coin \rightarrow VM_4$$
$$VM_4 = coffee \rightarrow VM_3$$

O processo VM_3 oferece o evento *coin*, aguarda a sincronização e se comporta com VM_4, este último por sua vez, oferece *coffee* e aguarda a sincronização, após a sincronização de comporta novamente como VM_3. Um outro processo equivalente a esses dois é VM_5,

$$VM_5 = coin \rightarrow coffee \rightarrow VM_5$$

que vai oferecer e sincronizar em *coin*, depois em *coffee*, e volta a se comportar recursivamente como ele próprio.

2.1.4 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 especificado 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. Considere como exemplo o processo VM_1 a seguir:

$$VM_1 = coin \rightarrow VM_2 \Box coffee \rightarrow VM_3$$

Esse processo oferece simultaneamente dois eventos: *coin* do lado esquerdo da escolha e *coffee* do lado direito. Fica a cargo do ambiente selecionar com qual dos eventos deseja sincronizar. Caso selecione sincronizar em *coin*, o processo VM_1 comunica *coin* e se comporta como o processo VM_2, caso contrário, se sincronizar em *coffee*, o processo VM_1 comunica *coffee* e se comporta como o processo VM_3. Escolha externa é também conhecida como 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.

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

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

$$ev_1 \rightarrow P \Box ev_2 \rightarrow P \Box \ldots \Box ev_x \rightarrow P$$

Onde os eventos ev_1, ev_2 e ev_x são todos eventos que pertencem ao conjunto C. Dessa versão indexada podemos definir o processo $R(C)$.

$$R(C) = \Box\, ev : C \bullet ev \rightarrow R(C)$$

O comportamento de $R(C)$ é sincronizar com qualquer evento que pertence ao conjunto C, e em seguida volta a comporta-se como $R(C)$ novamente, isto é, sincroniza em qualquer evento de C indefinidamente.

Já o operador de escolha interna (denotado pelo símbolo $\sqcap$) é utilizado quando se quer deixar a cargo 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 VM_1'

$$VM_1' = coin \rightarrow VM_2 \sqcap coffee \rightarrow VM_3$$

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

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

$$VM = coin \rightarrow VM_1 \Box coin \rightarrow VM_2$$

especifica um escolha externa entre dois eventos idênticos t. Dependendo da escolha de qual *coin* seja oferecido e sincronizado, o processo *VM* de comporta como *VM*_1 ou *VM*_2. Essa escolha será realizada internamente por *VM*, pois o ambiente não tem condições de escolher qual dos eventos *coin* será escolhido, pois são idênticos. Como conseqüência, *VM* é não-determinístico. O processo

$$VM' = coin \rightarrow VM_1 \sqcap coin \rightarrow VM_2$$

que emprega escolha interna, é equivalente a *VM* 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.

2.1.5 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 a escolha segue por um comportamento, caso contrário, quando falsa, a escolha segue o outro fluxo especificado. O construtor condicional básico pode ser visto no exemplo $\texttt{if}\ (g)\ \texttt{then}\ P(s)\ \texttt{else}\ P(s)$. 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*:

$$
\begin{array}{l}
COND(ev, C) = \\
\quad \texttt{if}\ (ev \in C)\ \texttt{then} \\
\quad\quad ev \rightarrow Skip \\
\quad \texttt{else} \\
\quad\quad ev \rightarrow Stop
\end{array}
$$

O processo *COND* possui dois parâmetros, *ev* um evento e C um conjunto de eventos. Caso o valor passado para *ev* pertença ao conjunto C, a expressão lógica $(ev \in C)$ 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 $\texttt{if}\ (g)\ \texttt{then}\ P\ \texttt{else}\ Stop$. O exemplo anterior pode ser abreviado como $(ev \in C)\ \&\ Skip$.

2.1.6 Internalização de Eventos

Quando queremos 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 processos. 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 Tabela 2.1. 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 *VM_5* apresentado na Seção 2.1.3. Escondendo de *VM_5* o conjunto de eventos $\{coffee\}$ temos o processo $VM_5 \setminus \{coffee\}$. O comportamento deste último é oferecer o evento *coin* para sincronização com o ambiente, após a sincronização, e por isso não precisa do aval do ambiente para ocorrer.

Através do operador de internalização é possível construir um processo cujo comportamento é equivalente ao processo *DIV*. O processo *DIV* representa um estado de *livelock*, 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.

Para criar um processo que se comporta como *DIV*, basta criar um processo recursivo que tenha todo o seu alfabeto escondido. Como exemplo, temos o processo *VMS* e *VM*:

$$VMS = coin \rightarrow VMS$$
$$VM = VMS \setminus \{coin\}$$

O processo *VM* se comporta como *VMS* quando os eventos de seu alfabeto $\alpha_{VMS} = \{coin\}$ são escondidos; funciona como a recursão infinita $P = P$, que é interpretada pelo ambiente externo como um *livelock*, ou divergência.

2.1.7 Composição Sequencial

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, como pode ser visto no exemplo $P(s)\ ;\ Q(t)$. O comportamento da composição consiste na "execução" do primeiro processo (que fica antes do símbolo ;) e que, quando termina com sucesso, se comporta como o segundo processo. Caso o primeiro não termine com sucesso (não tenha terminação ou termine como o processo *Stop*), o segundo processo nunca será executado. É importante ressaltar nesse exemplo que que o estado de P não é conhecido por Q. Isto é, qualquer mudança que ocorra a s internamente em P não é conhecido por Q. Isso também se aplica às leituras de canais realizadas dentro de P.

Por exemplo, o processo

$$a \rightarrow Skip;\ Q$$

que se comporta inicialmente como o processo $a \rightarrow Skip$, e após o término com sucesso (quando este se comporta como *Skip*) se comporta como o processo Q.

2.1.8 Interrupção

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

$$P \bigtriangleup Q$$

comporta-se como P até que o processo Q o interrompa, a partir de onde o processo se comporta como Q. A interrupção ocorre quando o ambiente sincroniza com qualquer evento que Q ofereça. Se P e Q oferecem os mesmos eventos ao ambiente, então ocorre uma escolha não-determinística entre eles.

Como exemplo do operador, temos o processo

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

que se comporta como $R(\{a,b\})$ até que $c \rightarrow Skip$ o interrompa. $R(\{a,b\})$ 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 $R(\{a,b\})$ é interrompido, e o processo se comporta como *Skip* (termina com sucesso).

2.1.9 Paralelismo

Até esse ponto, os processos se comportavam apenas sequencialmente: um processo é a continuação de outro (recursão na Seção 2.1.3), ou, o processo só inicia quando o outro anterior terminou com sucesso (composição sequencial na Seção 2.1.7). Através do paralelismo é possível executar mais de um processo simultaneamente, podendo haver (ou não) comunicação entre eles.

De uma forma geral, em todos os paralelismos apresentados que envolvem sincronização de eventos (síncrono, alfabetizado e generalizado), a sincronização ocorre de forma semelhante quando estão envolvidos dois processos (chamada de sincronismo simples ou *rendezvous*), ou quando estão envolvidos três 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 seus fluxos, 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 α_P e α_Q os alfabetos de P e Q, e P está em paralelo com Q

$$P \parallel Q$$

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 consequência, os processos comunicam evento a evento em parceria, caso contrário não comunicam. Na situação em que a intersecção dos alfabetos é vazia ($\alpha_P \cap \alpha_Q = \emptyset$), esse paralelismo se comporta como *Stop*.

De forma diferente, sejam X um conjunto tal que $X \subseteq \alpha_P$, e Y um conjunto tal que $Y \subseteq \alpha_Q$, 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 intersecção for aceito por apenas um dos processos, ele não poderá ocorrer

em nenhum dos dois, pois a sincronização é obrigatória. Só os eventos fora da intersecção dos alfabetos podem ser comunicados independentemente.

Uma outra variação do operador de paralelismo, chamada *interleaving*, permite compor processos em paralelo, sem que hajam 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 intersecção 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.

$$\begin{array}{lcl} P \parallel Q & \equiv & P\,|[\,\alpha_P \cup \alpha_Q\,]|\,Q \\ P\,{}_X\!\parallel_Y Q & \equiv & P\,|[\,X \cap Y\,]|\,Q \\ P \,|||\, Q & \equiv & P\,|[\,\{\}\,]|\,Q \end{array}$$

2.2 Semântica Algébrica e Operacional

Existem pelo menos três formas matemáticas distintas de entender sistemas especificados em CSP. São elas as semânticas algébrica, operacional e denotacional de CSP.

A semântica algébrica, como o próprio nome diz, é toda baseada em leis algébricas. 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, tornando-os mais simples ou atendendo a algum padrão estrutural, sem no entanto mudar o comportamento do processo original.

Outra representação, denominada 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 2.3). A semântica operacional de CSP é toda descrita a partir de sistemas de transições rotuladas (LTS – *Labeled Transition Systems*) [Kat05].

Formalmente, um LTS é uma quádrupla $M = (Q, A, \longrightarrow, q_0)$, em que Q é um conjunto finito não vazio de estados, $q_0 \in Q$ é o estado inicial, A é o alfabeto de ações e $\longrightarrow$ é a relação de transição tal que $\longrightarrow \subseteq Q \times A \times Q$ é um grafo direcionado [Jor95], cujas transições estão rotuladas pelos eventos que pertencem ao conjunto de ações A e seguem a relação de transição $\longrightarrow$.

A seguir iremos apresentar o modelo denotacional que não foi abordado nessa seção.

2.3 Semântica Denotacional e Refinamentos

A semântica denotacional de CSP emprega três modelos para estabelecer relações de refinamento entre processos. São eles os modelos de traces [3], de falhas e de 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. O conjunto de traces de um processo P, denotado por $traces(P)$, é o conjunto de todas as seqüências que P pode executar. 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 inclua este trace.
- $traces(Stop) = \{\langle\rangle\}$
- $traces(Skip) = \{\langle\rangle, \langle\checkmark\rangle\}$
- $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle\,\hat{}\,s \mid s \in traces(P)\}$ – Esse processo ou ainda não fez nada, ou seu primeiro evento foi a seguido de um trace de P.
- $traces(P \,\square\, Q) = traces(P) \cup traces(Q)$ – Esse processo oferece os traces de P e os traces de Q.
- $traces(P \sqcap Q) = traces(P) \cup traces(Q)$ – Uma vez que esse processo pode se comportar tanto como P quanto como Q, ele oferece tanto os traces de P e os traces de Q.
- $traces(\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q) = traces(P)$ se o valor de b for *true* ou $traces(Q)$ se o valor de b for *false*.

Entretanto, o modelo de traces não é capaz de identificar os eventos que um processo não pode executar num determinado instante. Assim, segundo esse modelo, o processo $P \,\square\, Q$ é equivalente ao processo $P \sqcap Q$, visto que ambos podem, eventualmente, ter o mesmo comportamento. O exemplo a seguir ajuda a entender melhor o problema. Seja o processo P equivalente $a \rightarrow Skip \,\square\, b \rightarrow Stop$, 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 $S \sqsubseteq_T I$ se:

[3] Apesar da palavra *trace* (ou seu plural *traces*) ser natural da língua inglesa, ao longo deste trabalho, quando utilizada, esta não será enfatizada, pois é muito comumente empregada e não existe um bom equivalente na língua portuguesa que transmita o mesma significado.

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

Isto é, 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.

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 irá recusar a executar naquele momento. O modelo de falhas inclui o modelo de traces. Assim, o modelo de falhas permite demonstrar que o processo $P \Box Q$ não é semanticamente 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, pois 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 comportamentos idênticos, se ambos possuem o mesmo modelo de falhas e divergências. Se P é equivalente a Q, então:

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

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 refinamentos só são válidos se os processos P e Q são equivalentes. Eles possuem exatamente os mesmos traces, falhas e divergências.

2.4 Suporte de Ferramentas

A ferramenta mais conhecida para prova de refinamentos sobre processos CSP é o FDR (*Failures-Divergences Refinement*) [Sys05]. O ProBE [Sys03] é outra ferramenta útil para animar modelos CSP. Ambas lêem especificações CSP descritas em uma linguagem funcional chamada CSP_M [Sca98], que é um acrônimo para *Machine Readable* CSP.

FDR é empregada dentro deste trabalho para verificar a validade dos refinamentos propostos no Capítulo 4. Já ProBE permite simular o comportamento passo a passo das especificações CSP que serão exemplificadas. É particularmente útil para o leitor que não está familiarizado com CSP e deseja ter um entendimento do funcionamento da especificação. Abaixo serão colocadas algumas definições relativas a notação CSP_M que serão úteis para entender os exemplos de especificação dos Capítulos 4 e 5.

Sintaxe do Processo CSP	**Sintaxe do Processo CSP$_M$**
Stop	`STOP`
Skip	`SKIP`
$a \rightarrow P$	a `->` P
$P(s)$	$P(s)$
$P(s) \Box P(s)$	$P(s)$ `[]` $P(s)$
$P(s) \sqcap P(s)$	$P(s)$ `|~|` $P(s)$
`if` (g) `then` $P(s)$ `else` $P(s)$	`if` (g) `then` $P(s)$ `else` $P(s)$
$P(s) \setminus C$	$P(s) \setminus C$
$P(s)\ ;\ P(s)$	$P(s)$ `;` $P(s)$
$P(s) \triangle P(s)$	$P(s)$ `/\` $P(s)$
$P(s) \|[C]\| P(s)$	$P(s)$ `[|`C`|]` $P(s)$

Tabela 2.2 Operadores de CSP$_M$ correspondentes aos operadores primitivos de CSP.

A Tabela 2.2 mostra na notação CSP$_M$, a relação dos operadores de processo CSP, inicialmente apresentados na Tabela 2.1.

Em acréscimo às definições da Tabela 2.2, temos outras construções CSP$_M$ que serão empregadas:

- `[]` x `:` a `@` p — escolha externa indexada;
- `|~|` x `:` a `@` p — escolha interna indexada;
- $s1$ `^` $s2$, `head`(s) e `tail`(s) — correspondem, respectivamente, aos operadores de sequência: 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,
- `#` s — retorna o número de elementos da sequência s.

CAPÍTULO 3

Testes de Software

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 **veredicto**. É uma tarefa de trabalho intensivo, podendo alcançar mais de 50% do custo total do desenvolvimento do software [GM04].

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, esse é chamado **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 está 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. Já **verificação**, objetiva assegurar consistência, completude e corretude do produto em cada fase e entre fases consecutivas do ciclo de vida do software. 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, o sistema sob testes será referenciado pela sigla SUT (*System Under Test*).

Um processo de teste consiste em um conjunto de atividades, papéis e artefatos para avaliar a qualidade do produto testado. As atividades mais essenciais são projetar e executar testes. Essas atividades são desempenhadas pelos papéis, 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. De acordo com o padrão IEEE [Com98], as seguintes definições são utilizadas para os artefatos produzidos pelo projetista de testes.

Item de teste (*Test item*) é o objeto a ser testado, por exemplo, um código-fonte, código objeto, dados de controle, etc.

Especificação de caso de teste (*Test case specification*) é 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*) é um documento especificando a seqüência de ações para a execução de um determinado caso de teste.

De forma mais comum na literatura, **teste** (artefato que o testador utiliza como entrada) é denominado como 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, se faz uso do termo **oráculo**, que é um mecanismo (humano – próprio testador, ou automático – uma ferramenta) que, considerando as entradas fornecidas no teste, saídas produzidas pelo SUT e saídas esperadas, define o veredicto da execução (passou/falhou). **Passar** significa que o SUT apresentou o comportamento especificado e **falhar** o caso contrário (o que caracteriza uma falha que deve ser corrigida).

Infelizmente, teste de software tem algumas desvantagens, como o seu alto custo, a impossibilidade de garantir a ausência completa de falhas no SUT e ainda pode ser foco de introdução de erros (o que pode tornar os resultados não confiáveis). Os erros introduzidos no teste podem acontecer da má especificação dos casos de teste e também da ausência de casos de teste para requisitos funcionais e não-funcionais.

Na literatura 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, estratégia de teste, etc. A seguir iremos introduzir a terminologia utilizada nesse trabalho.

3.1 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 3.1.

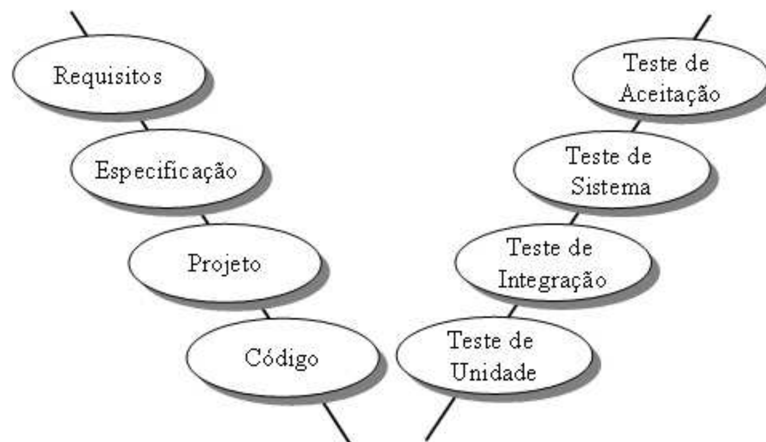


Figura 3.1 Modelo de desenvolvimento em V.

À esquerda da Figura 3.1 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 3.1, continuamos a avançar descendo fase por fase até alcançar a fase mais abaixo do processo de desenvolvimento que é onde o código é desenvolvido. 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 3.1. A seguir apresentamos a terminologia para cada um dos 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 é o nível de teste que após integrados os módulos e componentes, verifica o funcionamento do sistema dentro de um ambiente operacional de desenvolvimento.

Teste de aceitação verifica o funcionamento do sistema no ambiente operacional do cliente. Os testes de aceitação são subdivididos em alfa e beta, onde o alfa ainda não usa o ambiente do cliente e sim o da própria empresa. O destaque desse tipo de teste é o testador que passa a ser o próprio cliente. Já os testes de aceitação beta são executados nas máquinas do próprio cliente a fim de fazer a validação final do sistema.

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 testes de integração, a especificação está para os testes de sistema e os requisitos estão para os 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 (ou quase toda) a implementação está 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 de teste** consiste, para um 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 3.3 são descritas abordagens e tipos de teste.

3.2 Metas para um Processo de Teste

Devido ao seu 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 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 à cobertura alcançada, quantidade de falhas encontradas, etc.;

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

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 estão ligados à seleção de bons casos de teste (acompanhados de suas respectivas saídas) e à sua automação (que envolve controlar o processo de teste e observar os resultados dos testes). Na Seção 3.3 são mostradas algumas técnicas caixa-branca e caixa-preta empregadas para gerar casos de teste, tanto de maneira efetiva quanto eficiente (Metas 1 e 2). Na Seção 3.4 são mostradas ferramentas voltadas para automação de diversas atividades do processo de teste (Metas 3 e 4).

3.3 Abordagens

Existem basicamente duas **abordagens de teste**, uma denominada **caixa-preta** (*black box*) e outra denominada **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.

Considere o seguinte exemplo: seja um SUT 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 é uma técnica utilizada para identificar os principais limites de valores de dados onde mudanças significativas de comportamento do sistema testado podem ocorrer [Pre04]. 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 do SUT quando fornecidos valores próximos aos limites especificados.

Supondo que estamos testando o SUT 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 é uma técnica que seleciona casos de teste representativos considerando um grande número de casos de teste que possuem comportamento semelhante [Pre04]. A hipótese chave para sua aplicação é que o SUT 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 para cada partição como parâmetros de entrada fornecidos pelos casos de teste. Considerando que o SUT é o mesmo exemplo da técnica anterior, podem-se identificar três partições de valores: $[-100,0)$,[0], $(0,100]$; de onde se pode extrair (ao menos) um teste para cada uma das partições. Em princípio, esta é uma técnica 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 do SUT. Essa é uma técnica eficiente para a construção de casos de teste.

Na abordagem caixa-branca, o projetista de testes utiliza informações internas do software (estrutura) 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:

Cobertura de comandos (*statement coverage*) executa o código de uma maneira tal que cada comando da aplicação (*program statement*) é executado pelo menos uma vez.

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

Teste de mutação (*mutation testing*) objetiva executar o SUT 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*); nessa abordagem são levados em conta tanto aspectos comportamentais quanto aspectos 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. Para exemplificar a abordagem caixa-cinza, suponha que o desenvolvedor informe 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 à 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 esteja totalmente implementada. Já o de caixa-preta, baseia-se apenas na especificação, não podendo garantir que todas as partes da implementação são testadas. Portanto, caixa-preta consiste em testar o SUT contra a especificação para descobrir **faltas de omissão** (*omission faults*), indicando que partes da especificação não são cobertas. Já 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 e;
3. Comedida e cuidadosamente adicionar casos de teste até que o nível de cobertura esteja de acordo com alguma medida padrão da empresa.

De forma ortogonal às técnicas aqui descritas, existem **tipos de teste** que podem ser realizados independentes das classificações anteriores: testes de funcionalidade (verifica se a funcionalidade está como esperada), testes de performance (verifica o tempo de resposta do sistema), testes de carga (verifica a resposta do sistema dado a inserção variável de cargas), testes de interface (verifica a interface com o usuário), testes de estresse (testa a recuperação do sistema em situações críticas onde existe escassez ou ausência de recursos), testes de instalação (verifica se a instalação do software ocorre de maneira correta em diferentes ambientes operacionais), dentre outras [GM04, Jor95, Lew00].

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

3.4 Automação

A execução de casos de testes pode ser realizada manual ou automaticamente. A automação das atividades de teste visa tornar o processo mais ágil em atividades repetitivas, menos suscetível a erros humanos, mais fácil de reproduzir e menos dependente da interpretação humana [Que99]. A automação pode ser empregada em diversas atividades de teste, desde o planejamento dos testes, passando pela criação dos mesmos chegando até a sua execução.

Alguns tipos de teste, como os testes de performance e estresse, apenas podem ser executados de forma satisfatória quando automaticamente. Nesses tipos de teste, a repetição excessiva dos procedimentos pode levar o testador ao cansaço e aos erros que comprometerão os resultados finais.

Alguns requisitos de performance ou estresse requerem uma quantidade de interações impraticável para seres humanos, por exemplo, simular o acesso simultâneo de 1000 usuários ao

sistema. Ferramentas de automação da execução, como JMeter [Apa06], podem produzir a taxa de acesso que simula uma taxa de acesso impossível de ser obtida através de único ser humano. Além disso, este tipo de automação pode produzir métricas de qualidade e permitir a otimização 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 do SUT. Através de *scripts* é possível 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 do SUT. Isso pode melhorar a qualidade dos testes, com a garantia de que as mudanças de hardware não afetam o comportamento do SUT. Testes automáticos (executados automaticamente) permitem que a maior parte das funcionalidades básicas da interface do usuário possa ser coberta de forma que o SUT seja considerado maduro o suficiente para seguir para outros níveis de teste.

Um processo que emprega ferramentas de suporte à execução automática pode ser medido e repetido com maior facilidade, pois as ferramentas podem armazenar os resultados da execução em registros (*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 (criação de *scripts*) que tragam com rapidez o retorno do tempo investido para a criação dos *scripts* de automação.

Na Tabela 3.1 podemos ver uma pequena amostra das diversas ferramentas de teste existentes, cada uma com seu propósito específico.

Propósito da Automação	**Nome da Ferramenta**
Planejamento e projeto de testes	Rational TestManager [Sof99] e QADirector [Com06a]
Geração de massas de dados	File-Aid [Com06b]
API de testes de unidade e Verificação de assertivas	JUnit [JUn06] e Cactus [Jak06]
Testes de cobertura	PureCoverage [Rat06]
Inspeção e análise estática de código	JTest [Par06] e IntelliJ IDE [Int06]
Testes de GUI	Rational Robot [IBM06] e WinRunner [Mer06b]
Testes de carga e estresse	JMeter [Apa06] e LoadRunner [Mer06a]
Testes de desempenho e detecção de gargalos	JProbe [Sof06]

Tabela 3.1 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 indiscriminadamente, 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. Além disso tudo, testes não garantem ausência de erro e sim sua presença. Portanto, não é através de automação que resolveremos esse problema conceitual.

A adoção de um processo de automação de testes não é trivial. As organizações que pretendem introduzir o uso de ferramentas de automação precisam estar cientes das etapas necessárias e dos riscos envolvidos. Além do mais, o tempo gasto para automatizar um caso de teste precisa ser recompensado pelo tempo salvo pela sua execução automática, ou então a automação só trará prejuízos.

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

Em um estudo relatado em [Que99], considera-se 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% do esforço total destas atividades, quando se utiliza maciçamente ferramentas de automação em um processo que antes era totalmente manual. Podem-se destacar as atividades projeto de testes e de execução de testes como as que obtêm a redução mais significativa: 55% e 95%, respectivamente.

O fato é que os sistemas estão se tornando cada vez mais complexos, cada vez com mais componentes sendo desenvolvidos por diferentes fabricantes que utilizam diferentes técnicas e linguagens de programação, e ainda rodando em diferentes plataformas. Esses fatos motivaram a Motorola a desenvolver um *framework* de automação de testes chamado TAF (*Test Automation Framework*) [KKR+07, Rec06] que é usado para automação e execução de casos de teste escritos em alto nível para seus telefones celulares. A Seção 3.4.1 mostra mais detalhadamente o TAF, pois esse é o *framework* que utilizamos para a obtenção dos resultados apresentados no Capítulo 5.

O TAF que é utilizado internamente na Motorola difere dos demais *frameworks* de automação de testes pelo fato dos seus scripts de testes serem compostos por chamadas de métodos de alto nível de abstração. Assim, é possível manter os scripts de testes imutáveis para diversos produtos, enquanto os métodos (escritos no que chamamos de alto nível de abstração) são reimplementados somente quando necessário, para se ajustarem aos comportamentos específicos de cada telefone. Com isso é possível minimizar os custos com a manutenção dos casos de teste e aumentar o reuso dos métodos de alto nível. A seguir o TAF será mais detalhadamente explicado.

3.4.1 TAF – *Test Automation Framework*

TAF [KKR+07, Rec06] é um *framework* para automação de testes orientado a objetos projetado para suportar a automação de testes de integração e de sistema dos softwares embutidos em telefones celulares produzidos e desenvolvidos pela Motorola Industrial Ltda. O TAF é de-

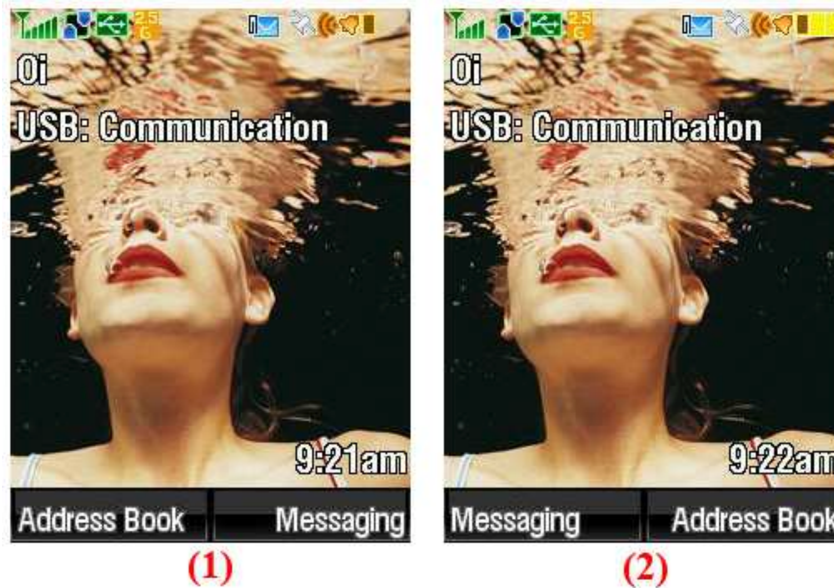


Figura 3.2 Telas.

senvolvido no *Brazil Test Center* (BTC), que é composto por uma rede de institutos de pesquisa e desenvolvimento sob liderança da Motorola. A principal motivação para o uso do TAF é a reutilização de código a partir da portabilidade de testes. Isto é, os testes que são executados em telefones de modelos diferentes, mas que implementam as mesmas funcionalidades, são os mesmos. Esses testes são escritos no que denominamos *alto nível* de abstração, ou seja, sem considerar detalhes específicos de implementação, hardware, posição de teclas, etc. Um exemplo desses detalhes pode ser observado na Figura 3.2. Note que a posição dos itens "*Address Book*" e "*Messaging*" nas telas (1) e (2) estão invertidos, porém através de chamadas às funcionalidades de alto nível do TAF, esses detalhes são abstraídos. Embora existam telefones que implementam as mesmas funcionalidades, a maneira como o usuário interage com estas funcionalidades varia em função do modelo do aparelho. Observando as funcionalidades comuns entre telefones diferentes e a elevada quantidade de testes em comum que é executada nestes aparelhos, projetou-se o TAF de forma tal que os mesmos casos de teste automatizados pudessem ser reutilizados em telefones diferentes. Além da reutilização dos casos de teste, a automação destes também reduz o tempo gasto no ciclo de desenvolvimento do software embutido nos telefones, eliminando a necessidade da execução manual de testes.

Para interagir com o telefone, o TAF utiliza uma biblioteca proprietária de funções chamada PTF (*Phone Test Framework*) [EV06]. O PTF comunica-se com o telefone via interface USB (*Universal Serial Bus*) para prover funções que simulam o pressionamento de teclas e retornar o texto que está sendo mostrado na tela do telefone. Com o PTF é possível estimular o teclado do telefone e verificar se o conteúdo mostrado na tela do aparelho é o esperado. As funções providas pelo PTF, entretanto, são de baixo nível de abstração; a utilização direta do PTF para automatizar um caso de teste produz *scripts* de teste ilegíveis e difíceis de serem mantidos e portados para serem executados em outros modelos de telefone. Apesar disso, o PTF representa uma base bastante apropriada para implementação de testes automáticos.

TAF Test Script	PTF Test Script
// Launch messaging application *navigationTk.launchApp(MESSAGING);*	*// Go to Idle (initial screen)* *ptf.nav.idle();* *// Enter in Main Menu* *ptf.phone.pressKey(RIGHT);* *// Scroll to Messages* *ptf.nav.scrollTo(MESSAGES);* *// Select to enter in Messages* *ptf.phone.pressKey(CENTER);*

Tabela 3.2 Script de teste TAF *versus* PTF.

O TAF foi projetado para resolver o problema do baixo nível de abstração das funções providas pelo PTF e promover a reutilização de código de testes automatizados. No TAF, a seqüência de teclas a serem pressionadas ou a verificação do conteúdo mostrado na tela do telefone (isto é, a chamada direta a métodos do PTF) são encapsulados em *Utility Functions* (UFs). UFs são entidades primitivas que isolam hierarquicamente as funcionalidades da implementação. Assim, uma UF é a implementação de um passo de alto nível que é executado em um caso de teste. Com o TAF, os casos de teste são escritos em termos de UFs de alto nível, ao invés de chamadas a funções de baixo nível, como as que o PTF provê. A Tabela 3.2 representa um passo de um mesmo script de teste no TAF e no PTF. Observe que o TAF faz chamada a um único comando enquanto o PTF, para realizar a mesma tarefa, executa quatro comandos. Nesse caso, a UF *LaunchApp* abstrai detalhes específicos de navegação para atingir o seu objetivo, enquanto o script em PTF precisa explicitar todos os passos (vai para a tela inicial do telefone *idle*, abre o menu principal, navega até o item desejado e o seleciona pressionando "*center*". Alguns exemplos de UFs que representam um passo de alto nível em um caso de teste são: *Call* (faz uma chamada para o número especificado), *LaunchApp* (abre uma aplicação, como o editor de mensagens ou a lista de contatos), *CapturePictureFromCamera* (fotografa utilizando a câmera do telefone), *DeleteFile* (remove um arquivo qualquer do telefone, como uma figura, som ou vídeo), entre outras.

A Figura 3.3 mostra um exemplo de caso de teste de alto nível automatizado utilizando o TAF. Cada passo do caso de teste tem um procedimento (.P) e um resultado esperado (.ER) associado. Este fragmento executa a seqüência de passos descritos na Tabela 3.3[1]. Note que cinco UFs são empregadas no fragmento de caso apresentado na Figura 3.3: *LaunchApp* (nesse caso, esta UF abre a aplicação de IM – *Instant Messaging* – que provê a comunicação entre usuários através de um chat), *CheckScreen* (verifica se a tela desejada está sendo mostrada), *LoginIM* (faz o login na aplicação de IM), *ScrollToContact* (navega através dos itens de uma lista e seleciona um contato específico nessa lista), e *ExitIM* (que simplesmente sai da aplicação de IM).

Embora telefones diferentes exibam comportamentos distintos de entrada e saída, um mesmo caso de teste de alto nível é aplicável para vários modelos de telefone de uma determinada

[1]O caso de teste está escrito em inglês (Linguagem Natural Controlada – LNC) para uma melhor associação de cada passo com o seu correspondente de um caso de teste automatizado utilizando o TAF.

Step	Procedure	Expected Results
1	Start IM application	IM application is started
2	Login	Contact List is shown
3	Select a contact	Conversation is opened
4	Exit IM application	Phone goes to idle

Tabela 3.3 Exemplo de caso de teste escrito em inglês (LNC).

```
// Step 1.P: Start IM application
navigationTk.launchApp(PhoneApplication.IM);

// Step 1.ER: IM application is started
phoneTk.checkScreen(PhoneScreen.IM);

// Step 2.P: Login
imTk.loginIM();

// Step 2.ER: Contact List is shown
phoneTk.checkScreen(IMScreen.CONTACT_LIST);

// Step 3.P: Select a contact
imTk.scrollToContact(IMContact.JOHN);

// Step 3.ER: Conversation is opened
phoneTk.checkScreen(IMScreen.CONVERSATION);

// Step 4.P: Exit IM application
imTk.exitIM();

// Step 4.ER: Phone goes to idle
phoneTk.checkScreen(PhoneScreen.IDLE);
```

Figura 3.3 Um fragmento de um caso de teste em TAF.

família, desde que eles tenham as mesmas *features* implementadas. Conseqüentemente, um caso de teste é automatizado somente uma vez para todos os produtos de uma mesma família (ou seja, um caso de teste em TAF pode ser executado em qualquer telefone de uma mesma família). Porém não necessariamente todos os telefones de uma mesma família apresentam exatamente o mesmo comportamento. Caso esse comportamento seja diferente, é necessário adaptar as UFs para considerar esse novo comportamento. Esse procedimento de adaptação das UFs para executar em produtos com comportamento distinto é chamado *porting*. A seguir será descrito como o TAF foi projetado para permitir o porting eficiente de casos de teste escritos em alto nível.

Para permitir o reuso de casos de testes automatizados, o TAF foi projetado para resolver as limitações introduzidas pelas APIs (*Application Programming Interfaces*) do PTF escritas em baixo nível. Podemos considerar limitações do PTF a criação de scripts difíceis de entender e de manter. A Figura 3.4 apresenta uma visão simplificada da organização do TAF. Como pode

ser visto na Figura 3.4, UFs são classes que implementam uma interface chamada *Step*. Sob certo ponto de vista, um caso de teste é uma lista de *Steps*; tudo o que um caso de teste faz é invocar o método *execute()* que é definido na interface *Step* e implementado nas UFs concretas. Cada UF, entretanto, além de implementar a interface *Step*, possui uma API bem definida que provê métodos para possibilitar a interação com a UF em questão. Por exemplo, as UFs *LaunchApp* e *ScrollToContact* definem respectivamente os métodos *setApplication()* e *setContact()*. O primeiro método indica à *LaunchApp* qual aplicação deverá ser iniciada. Já o segundo, define qual contato deverá ser selecionado quando a UF *ScrollToContact* for executada. A classe *BaseTestCase* armazena uma coleção de objetos *Step*. Assim, *BaseTestCase* pode ser vista como a classe pai de todos os casos de teste (e.g. TC_01). As implementações das UFs invocam APIs do PTF e representam o comportamento baixo nível de cada telefone (e.g. *LaunchAppImp* e *ScrollToContactImp*). Essas implementações estão organizadas em pacotes distintos (e.g. *Common Implementation* e *Phone X*). Na Figura 3.4, assuma que X representa um novo modelo de telefone. Dado um caso de teste pré-existente, vamos analisar como o TAF suporta o porting de um caso de teste para o telefone X. Inicialmente, deve ser verificado se a execução de cada UF que compõe o caso de teste que está sendo portado corresponde ao comportamento esperado para o telefone em questão. Caso alguma UF não esteja apresentando um comportamento adequado, uma nova implementação para essa UF deve ser criada. Por exemplo, uma implementação específica da UF *ScrollToContact* foi criada para o telefone denominado *Phone X*. Uma vez que as UFs são estendidas somente se já não exista uma implementação de UF compatível, o TAF maximiza a quantidade de reuso de software.

Para permitir a instanciação correta das implementações para um telefone, o TAF implementa a noção de *Feature Toolkit* (e.g. *Phone Toolkit*, *Navigation Toolkit*, *IM Toolkit*), como ilustrado no final da Figura 3.4. Uma vez que o TAF tem potencialmente mais de uma implementação para cada UF (API), os *Feature Toolkits* devem saber a implementação apropriada para cada UF que será executada no telefone sob teste. Essa informação é codificada em arquivos XML, como apresentado na Figura 3.5, os quais são mantidos pelos desenvolvedores do TAF. Outro papel dos *Feature Toolkits* é a adição das UFs instanciadas em uma lista de passos a serem executados e também de iniciar a execução desses passos. Além disso, eles também são responsáveis por informar às UFs os seus respectivos argumentos, como por exemplo, chamar do método *setApplication(App)* da API *LaunchApp* a fim de que esta UF seja informada de qual aplicação (*App*) deve ser iniciada. Tão logo a lista de passos é criada pelos *Feature Toolkits*, a execução do caso de teste é iniciada. Brevemente, um caso de teste em TAF consiste de várias chamadas de métodos encapsulados em *Feature Toolkits*, como pode ser visto na Figura 3.3 (são chamados métodos do *Navigation Toolkit*, *Phone Toolkit* e *IM Toolkit*).

A Figura 3.6 mostra a arquitetura em camadas utilizada pelo TAF, onde é possível visualizar a relação entre a camada onde estão os casos de teste, a camada onde os *feature toolkits* são implementados, a camada das UFs (API + implementações), e o PTF.

Considerando o TAF, praticamente todo o tempo gasto para o desenvolvimento de novos casos de testes está relacionado a criação ou modificação de UFs. Para criar novos casos de teste, os desenvolvedores fazem reuso das funcionalidades já implementadas no TAF e, se necessário, novas UFs são implementadas. Para portar casos de testes já existentes para novos produtos, o processo é diferente. Como dito anteriormente, os casos de teste são escritos em

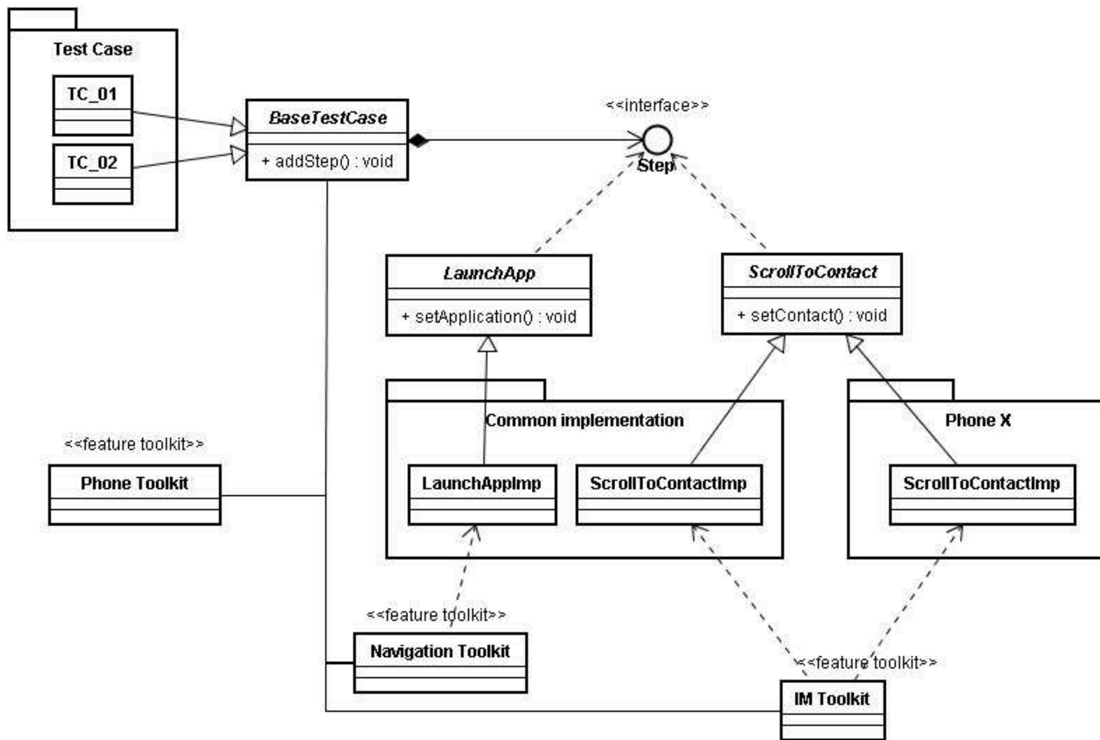


Figura 3.4 Diagrama de classes do TAF.

```
<family_definition name="FAMILY_01">

  <common_mappings>
    <uf api="taf.uf.im.api.ScrollToContact"
        imp="taf.uf.im.common.ScrollToContactImp"/>
  </common_mappings>

  <phone name="PhoneX">
    <mappings>
      <uf api="taf.uf.im.api.ScrollToContact"
          imp="taf.uf.im.common.phonex.ScrollToContactImp"/>
    </mappings>
  </phone>

</family_definition>
```

Figura 3.5 Exemplo de mapeamentos de UFs em arquivo XML.

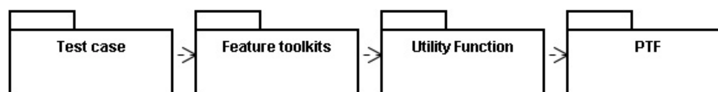


Figura 3.6 Arquitetura de camadas do TAF.

alto nível e esses casos de testes mantém-se imutáveis para todos os produtos de uma mesma família. Assim, para portar casos de teste, os desenvolvedores buscam as diferenças dos produtos (caso elas existam) e estendem as UFs (criando assim implementações específicas) para resolver as diferenças e fazer com que o teste esteja apto a ser executado no novo produto.

Atualmente não existe nenhum trabalho em nosso grupo de pesquisas com a finalidade de desenvolver novas *Utility Functions* para o TAF de forma automática. Um desafio antigo envolvido no TAF é a possibilidade de gerar novas UFs automaticamente para minimizar o tempo gasto para criar novos caso de teste em TAF. O presente trabalho propõe a geração de código das UFs automaticamente baseado em modelos CSP, que são representações do comportamento dos telefones, extraídos também de forma automática. Tendo em vista a relevância em poder gerar automaticamente as UFs, o capítulo anterior apresentou uma notação formal apropriada para descrever comportamento de um SUT que será usada nos capítulos subseqüentes para gerar UFs automaticamente.

CAPÍTULO 4

Geração de Código para um Framework de Automação de Testes

A execução de testes automáticos ajuda a minimizar esforço de testadores, verificar padrões e melhorar a qualidade do produto sob teste. Por outro lado, a automação de testes é onerosa e precisa ser cuidadosamente estimada para não trazer perdas às organizações [dOGF06]. Algumas razões fazem com que a aplicação de um processo de automação de testes seja um investimento atrativo para empresas que trabalham com desenvolvimento de software. Por exemplo, durante o ciclo de vida da execução manual de casos de teste, os resultados podem ser alterados devido a erros humanos. O cansaço e/ou o estado emocional do testador podem ser os causadores da execução ou interpretação errada de um caso de teste, podendo esse ter o seu veredicto alterado indevidamente. Considerando esse aspecto, quando um caso de teste está automatizado ele sempre executará exatamente os passos que devem ser executados e nunca terá o resultado do teste alterado pelos fatores humanos supracitados. Além disso, casos de teste automatizados podem rodar em paralelo, sem interrupções e seus resultados podem ser facilmente analisados [Que99].

Considerando o *framework* de automação de testes utilizado na Motorola (TAF), praticamente todo o tempo gasto no desenvolvimento de casos de teste está relacionado com a criação de novas funcionalidades para o *framework*. O principal esforço nessa atividade é o de desenvolver novos casos de teste e o de portar[1] casos de teste já existentes para novos produtos. No processo de criação de novos casos de teste, os desenvolvedores reusam as UFs já existentes no TAF e, se necessário, criam novas UFs. Um caso de teste em TAF, nada mais é do que uma composição de chamadas a UFs, como descrito no Capítulo 3. Para portar casos de testes já existentes para novos produtos, o processo é um pouco diferente. Como dito anteriormente, os casos de teste em TAF são escritos através da chamada de métodos de alto nível de abstração e cada método desse executa uma determinada UF. Foi mencionado também que os casos de teste são aplicáveis para todos os produtos de uma mesma família e, algumas vezes, para produtos, inclusive, que rodam em diferentes plataformas. Assim, para portar um caso de teste, os desenvolvedores executam um teste já existente no telefone alvo e encontram as diferenças de implementação entre os produtos (se existirem). Após encontrar as diferenças na execução das UFs, essas últimas são estendidas para que se adequem ao comportamento do novo produto sob teste.

[1] O termo portar é um jargão utilizado pela equipe de desenvolvimento do TAF para a atividade de adequar um caso de teste já existente para novos produtos (inclusive produtos de diferentes plataformas). Como o TAF é um *framework* em que os casos de teste são escritos em alto nível, um mesmo caso de teste pode ser aplicado a uma gama de produtos, inclusive produtos que rodam em plataformas diferentes.

Até então não existia nenhum trabalho dentro do grupo de pesquisa do BTC com o objetivo de desenvolver *Utility Functions* para o TAF automaticamente e essa seria uma boa oportunidade de contribuição. Decidimos então resolver esse problema tomando como base dois trabalhos existentes no grupo de pesquisa. O primeiro objetiva a geração de casos de teste escritos em linguagem natural controlada (LNC), baseado em documentos de requisitos também escritos em LNC [CS06, dCN06, Car06, Lei06]. Essa geração é baseada no processamento de uma representação intermediária (modelo CSP) dos requisitos. A saída desse processamento é um conjunto de testes representados em LNC (e cada caso de teste tem uma representação intermediária em CSP). O segundo trabalho tem por objetivo a geração de casos de testes automatizados para o TAF para cada um dos casos de teste gerados pelo primeiro trabalho [dS07]. Nesse trabalho o autor usa a representação intermediária de cada caso de teste para obter scripts de teste automatizados em TAF. A exemplo do primeiro trabalho, uma representação CSP de cada caso de teste gerado para o TAF também é mantida. Mostraremos a seguir que esse último trabalho visa gerar somente o código para a camada de casos de teste do TAF. Isto é, para cada funcionalidade não encontrada na base de dados do TAF, o autor representa essa funcionalidade com um evento especial denominado *prototype*, o mesmo utilizado na Motorola quando há necessidade de criar a implementação para uma determinada UF do TAF. Porém não fazia sentido escrever os requisitos em LNC, derivar casos de teste, codificar esses casos de teste em scripts TAF e não poder executá-los caso alguma funcionalidade necessária não estivesse presente no TAF. O trabalho aqui proposto tem por objetivo preencher essa lacuna entre a geração automática de casos de teste para o TAF e sua execução. Um desejo antigo dos gerentes das equipes de automação de testes da Motorola era justamente esse: desenvolver UFs para o TAF automaticamente. Nessa dissertação é apresentada uma abordagem para desenvolver automaticamente essas UFs baseado em modelos CSP (que são representações do comportamento dos telefones).

Quando a automação de testes com o TAF foi iniciada, o maior desafio era criar uma arquitetura que permitisse o reuso e a extensibilidade (a Seção 3.4.1 mostra os princípios de arquitetura do TAF). Uma vez que esses princípios foram implementados, o TAF pôde então ser visto como uma camada de abstração do PTF. Nós observamos que o código das UFs é composto de chamadas de métodos do PTF e de algumas chamadas de métodos do próprio TAF. Basicamente, para cada tela, o PTF provê um conjunto de nomes, posição, estado, etc. de cada item da tela. Depois de analisar o código de várias UFs, nós verificamos alguns padrões de código que são utilizados em praticamente todas as UFs. A partir daí, iniciamos uma análise mais profunda com o objetivo de verificar a viabilidade da criação de novas UFs automaticamente. O resultado dessa análise gerou a idéia do trabalho aqui apresentado.

O PTF pode simular o pressionamento de teclas, pegar informações da tela e checar se a informação mostrada na tela é a esperada pelo caso de teste. Atualmente duas opções para capturar e analisar as informações da tela estão disponíveis pela API do PTF: uma é baseada na captura do conteúdo lógico da tela e a outra provê o conteúdo da tela graficamente. A Figura 4.1 mostra um exemplo das duas formas de capturar o conteúdo da tela dos celulares. Note que para cada item gráfico da tela o PTF retorna um conjunto de propriedades. Essas informações são a base para a codificação de uma UF.

Nós criamos uma ferramenta chamada *Behavior Extractor Tool* (BxT) [NB07] –Seção 4.2–

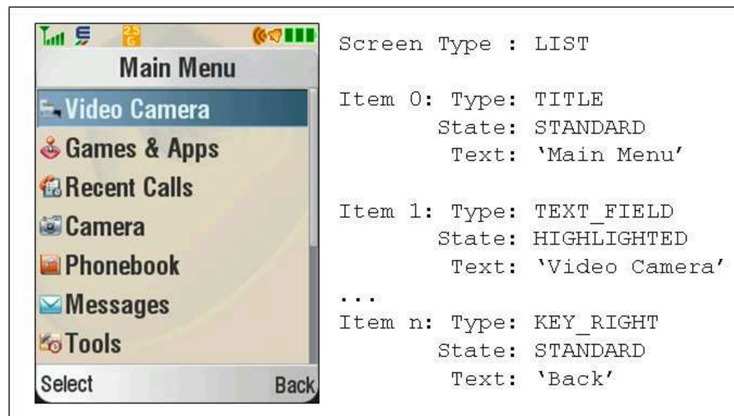


Figura 4.1 Exemplo de captura de conteúdo gráfico e lógico do PTF para a tela do menu principal de um telefone celular.

que extrai o comportamento do telefone para uma especificação CSP, utilizando o PTF para isso. Associada a essa especificação, a BxT também armazena o conjunto de itens de cada tela e suas transições (por exemplo, a tecla –ou conjunto de teclas– que foram pressionadas para chegar a uma determinada aplicação ou realizar uma determinada ação no telefone).

Uma vez que nós temos o modelo formal da aplicação do telefone (que denominamos `M`) e o modelo formal dos casos de teste (que denominamos `TC`) representados em CSP, nós podemos verificar se `TC` é parte de `M` através de refinamento. É importante salientar que nós assumimos que o modelo `M` extraído do telefone celular possui o comportamento esperado para o caso de teste `TC`. Nós criamos um algoritmo para encontrar todos os sub-traces que representam em `M` cada evento *prototype* do `TC`. Depois de descobrir todos os sub-traces, nós temos todas as informações sobre as telas e transições de cada evento *prototype*. Finalmente, nós geramos o código das UFs baseado nas informações de cada tela e das transições (ordem de pressionamento de teclas) que o PTF e está armazenado junto com o modelo.

Nesse capítulo será apresentada a abordagem proposta para realizar a geração de código de forma automática para o *framework* de automação de testes utilizado na Motorola. O código gerado é baseado em modelos comportamentais dos telefones celulares da Motorola, sendo esses modelos também gerados (extraídos) automaticamente. Os modelos comportamentais extraídos representam a forma que o telefone celular se comporta, ou seja, define como o celular age ao pressionarmos teclas, que aplicações podem ser acessadas a partir de uma outra aplicação, etc. Esses modelos são representados como especificações CSP e durante sua extração, todos os itens relacionados a GUI (*Graphical User Interface*) também são armazenados em uma estrutura de dados, bem como as teclas que foram pressionadas para que aquela tela pudesse ser alcançada. Os itens da GUI junto com a informação de como chegar em cada um desses itens são essenciais para produzir o código-fonte das UFs, pois uma UF nada mais é do que a execução do sistema através de uma visão caixa preta do mesmo.

A abordagem proposta para a geração de código automática de UFs para o TAF consiste de quatro passos que serão listados a seguir:

1. Obter o script de teste prototipado;

2. Extrair o modelo comportamental;

3. Executar o algoritmo proposto para obter uma representação em CSP das UFs que precisam ser desenvolvidas;

4. Implementar as UFs identificadas.

Cada um desses itens será explicado mais detalhadamente nas seções seguintes.

4.1 Obtenção do Script de Teste Prototipado

Uma das áreas de interesse do grupo de pesquisa do BTC é o de *geração de modelos de uso baseada em documentos de requisitos* [CS06]. Esta é a primeira atividade no processo de geração de casos de teste. Nesse momento, as funcionalidades do sistema estão especificadas em casos de uso escritos em LNC. O principal objetivo da padronização da escrita de casos de uso e casos de teste é a redução da ambiguidade no texto, melhorando assim a qualidade dos documentos e minimizando o tempo de revisão. Além disso, os requisitos e casos de teste escritos em LNC facilitam o seu processamento automático. Uma vez que temos as funcionalidades do sistema escritas em LNC, é possível processar automaticamente essa documentação a fim de gerar um modelo formal de uso do sistema (como um todo). Esse modelo é uma representação formal de todas as possíveis ações do usuário sobre o sistema (nesse caso representado em CSP).

Outra atividade do grupo é a *geração de casos de teste*. Essa atividade representa o objetivo geral do grupo de pesquisas que é justamente a melhoria do processo de testes através da automação de todas as atividades envolvidas no processo. Para que a geração de casos de teste ocorra, o modelo de uso em CSP do sistema sob teste é dado como entrada. A geração é guiada por propósitos de teste [dCN06, Car06] que indica quais funcionalidades devem ser consideradas na geração utilizando algum critério de cobertura dos requisitos. Os benefícios diretos dessa abordagem são o aumento da eficiência e da produtividade, bem como a melhoria na qualidade dos casos de teste gerados.

Em [dS07], o autor usa esses casos de testes gerados (denominados casos de teste abstratos) como entrada para que cada um deles seja traduzido em scripts de teste para o TAF. É importante salientar que nesse trabalho cada caso de teste automatizado é representado tanto em CSP (representação intermediária não disponível para o usuário final) quanto em código Java (que representa o script de teste para o TAF como ilustrado na Figura 4.4). É importante salientar que para cada funcionalidade não encontrada no TAF, a abordagem proposta em [dS07] é representar essa funcionalidade como um evento especial chamado *prototype*. Esse evento é fundamental para a obtenção dos resultados apresentados neste capítulo e no Capítulo 5. Esse evento representa as UFs do TAF que precisam ser implementadas para que o caso de teste (até então) parcialmente implementado possa ser executado. Para que isso seja possível, nós propomos uma possível implementação para cada evento *prototype* encontrado na especificação. Nesse aspecto, o presente trabalho pode ser visto como uma extensão do trabalho [dS07]. Nossa principal contribuição é a criação de uma implementação executável para os casos de teste que são gerados automaticamente através do processamento da LNC que define os docu-

mentos de requisitos. A seguir o trabalho de Souza [dS07] será mais detalhadamente abordado para que essa dissertação fique auto-contida.

4.1.1 Modelagem de Casos de Teste como Modelos Formais Abstratos

Antes de apresentarmos a abordagem utilizada para modelar casos de testes como modelos formais, é necessário introduzir algumas definições para que o leitor entenda mais facilmente a terminologia. Um desses conceitos é o de Testes Baseados em Modelos (*Model-Based Testing*), ou simplesmente MBT. MBT está relacionado com a geração automática de procedimentos de teste usando modelos do sistema [DJK+99]. A idéia do MBT é ter um modelo (especificação) do sistema e usar tal modelo para gerar sequências de entradas e saídas esperadas (procedimentos de teste). A entrada é aplicada ao sistema sob teste e a saída do sistema é comparada com a saída do modelo (visão caixa-preta). A saída do modelo pode ser verificada através do resultado da execução do modelo (traces resultantes), após executar os procedimentos de entrada. Esse procedimento valida o modelo, ou seja, indica que o modelo representa o comportamento do sistema.

Outro conceito importante é o de Anti-MBT, ou *Anti-Model-Based Testing* [BIMP04]. Anti-MBT é uma abordagem para criar modelos abstratos (modelos de teste) do sistema usando o processo de engenharia reversa. Esse conceito é útil quando não é possível ter um modelo formal abstrato definido *a-priori*. Assim, a abordagem Anti-MBT gera o modelo formal do sistema seguindo os traces de sua execução que por sua vez são descritos através da execução de casos de teste. As execuções do sistema fornecem os comportamentos necessários para construir um modelo abstrato do Sistema sob teste (SUT – *System Under Test*). Assim, depois da obtenção de um modelo abstrato, MBT pode ser aplicado como usual. Na abordagem proposta em [dS07], foram aplicadas técnicas similares à Anti-MBT para criação de uma especificação formal do sistema.

Casos de teste de sistema (tanto manuais quanto automáticos) descrevem o comportamento do sistema através de cenários de teste. Souza [dS07] propôs então aplicar engenharia reversa dos casos de teste de sistema para obter uma representação formal do SUT baseado nos modelos extraídos. Entretanto, antes de criar um modelo do sistema inteiro, o primeiro passo é modelar cada caso de teste como um modelo formal individual.

A estratégia de utilizar os próprios casos de teste de sistema para gerar um modelo abstrato do SUT é interessante porque casos de teste significam comportamentos reais (ações) do SUT. Os casos de teste são especificados e executados diretamente usando o sistema. Além disso, em um processo de teste convencional, os casos de teste são continuamente criados e modificados para se adaptarem para novos comportamentos ou modificações do SUT. Os projetistas de teste, que por sua vez também são testadores, especificam e/ou modificam casos de teste continuamente. Assim, foi observado que, em situações reais, várias informações atualizadas sobre o SUT são obtidas através da experiência dos testadores. Os projetistas conhecem bem as funcionalidades sob teste, e assim eles adicionam novas informações quando estão especificando novos casos de teste ou modificando casos de teste já existentes. Usualmente, tal informação não estão contidas nos documentos de requisitos que serviram como base para a criação desses casos de teste.

Usar casos de teste para construir modelos do sistema é a forma mais fácil de extrair o

comportamento do sistema automaticamente. Uma vez que vários casos de teste estão especificados seguindo um fluxo sequencial, a extração de cada cenário de teste se transforma em uma tarefa fácil. O mesmo não ocorre quando temos outros tipos de documento que descrevem o comportamento do sistema de forma complexa, usando diferentes cenários e sequências. Nessa situação, é difícil extrair o comportamento do sistema corretamente. Na abordagem proposta no presente trabalho, um modelo formal é extraído a partir do próprio SUT, pois informações concernentes à execução do sistema são de fundamental importância para a geração de UFs.

4.1.2 Modelagem de Casos de Testes Formais

Para realizar a modelagem de casos de testes, um conjunto de scripts de teste é dado como entrada e os modelos abstratos correspondentes a cada script de teste é disponibilizado como saída. Além de gerar modelos abstratos, a abordagem também executa o mapeamento entre os eventos CSP gerados através do processamento de LNC e os comandos de script correspondentes. Uma vez que um mapeamento é armazenado em um banco de dados, o mesmo pode ser usado para gerar scripts de teste concretos em TAF. A abordagem MBT usa duas representações para os casos de teste: uma representação abstrata e uma concreta. Na abordagem proposta por Souza, casos de teste abstratos (também conhecidos como casos de testes formais) são representados na notação CSP ou LTS. Para o presente trabalho, somente a parte CSP dessa representação interessa. Diferentemente dos casos de teste abstratos, casos de teste concretos podem ser executados (manual ou automaticamente) seguindo os passos definidos no script. A Figura 4.2 nos dá uma visão geral da abordagem utilizada para a modelagem de casos de teste.

De acordo com a Figura 4.2, o primeiro passo da abordagem é fazer um *parse* de todos os scripts de teste dados como entrada. Esse passo é responsável por ler cada script de teste e extrair todo o conteúdo do teste (*setup*, *steps* e *cleanup*). Uma vez que todas as informações são extraídas, o conteúdo dos comandos ***description*** (frases em inglês) são enviadas ao sistema de processamento de linguagem natural [Lei06]. Como mostrado anteriormente, o comando *description* descreve o objetivo da sequência de ações, escritas usando uma linguagem de programação, em alto nível (ver Figura 4.3). O resultado do processamento de uma frase em inglês é um evento CSP correspondente. Todos os eventos CSP traduzidos são enviados aos módulos de geração e mapeamento. Finalmente, depois de executar o passo de modelagem, modelos abstratos (CSP ou LTS) e um conjunto de mapeamentos entre eventos CSP e comandos de script são gerados.

4.1.2.1 Processamento de Linguagem Natural

Como mostrado na Figura 4.2, o passo de processamento de linguagem natural é muito importante para a modelagem utilizada em [dS07], bem como para o presente trabalho. O processamento de linguagem natural tem por objetivo traduzir frases escritas em linguagem natural (inglês) em eventos CSP ou transições LTS. Consequentemente, para que isso seja possível é necessário definir as estruturas usadas para realizar a tradução. Tais estruturas são termos e classes gramaticais, bancos de dados de conhecimento, e *parsers* semânticos e sintáticos.

O trabalho apresentado em [Lei06] mostra uma estratégia para o processamento de casos de teste manuais especificados em inglês e os traduz em casos de teste para a notação CSP.

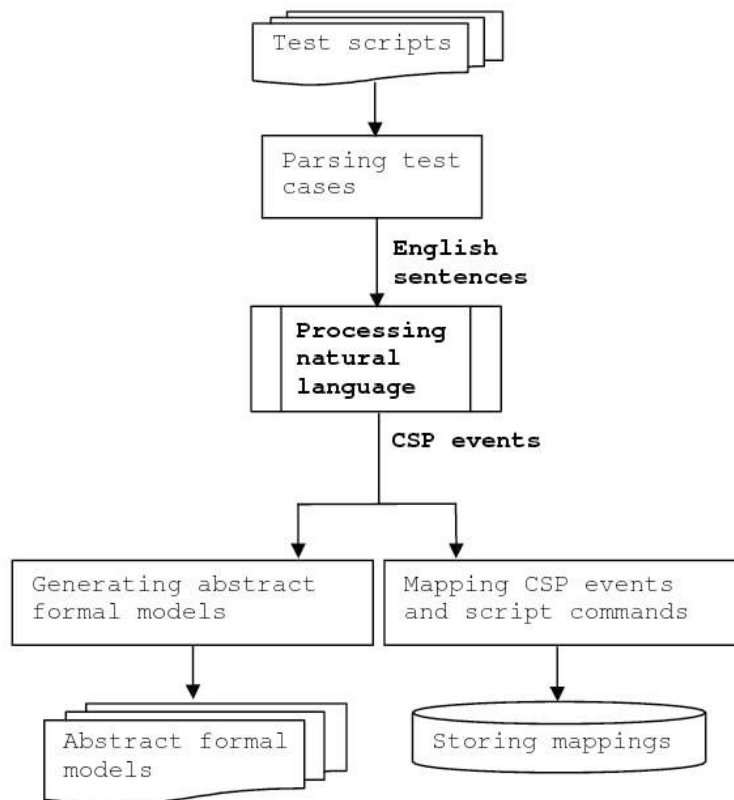


Figura 4.2 Abordagem utilizada para modelagem de casos de teste.

Esse trabalho usa uma estratégia baseada em ontologia de domínio. Em outras palavras, os termos contêm valores semânticos definidos por uma ontologia específica. Tomando a ontologia como base, ele utiliza *parsers* semânticos e sintáticos para capturar e interpretar as frases. Como a aplicação de domínio que estamos utilizando é a de sistemas de telefonia celular, o valor semântico dos termos definidos no banco de dados de conhecimento são estritamente relacionados a esse domínio. Assim, termos como "phone", "message", "SMS", "phonebook", "contact number" são frequentemente apresentados.

4.1.2.1.1 Definição do alfabeto CSP: CSPHeader Para a obtenção de casos de teste abstratos (na notação CSP) é necessário especificar um conjunto de definições CSP (*datatypes*, *channels*, tuplas, entre outros) capazes de reconhecer e manipular um evento CSP. Como mostrado anteriormente, cada evento CSP (abstrato) corresponde a uma ação real e essa relação é armazenada na tabela de mapeamentos.

Ações reais são descritas em linguagem natural (inglês) e especificam o que os testadores devem fazer em um caso de teste. Por exemplo, o evento

`delete.DTDEL_ITEM.(MESSAGES, {ALL})`

apresentado em casos de teste abstratos significam a ação real *Delete all messages* em um caso de teste concreto, como pode ser visto na Figura 4.3. Nesse exemplo, o passo que deleta todas as mensagens foi especificado por um canal `delete`, que significa a ação de deletar, e um

Figura 4.3 Objetivo em alto nível *versus* sequência de ações.

elemento (dado comunicado) de um *datatype*: `DTDEL_ITEM.(MESSAGES, {ALL})`. Este dado comunicado através do canal `delete` é interpretado como: todos os itens de mensagem do telefone celular devem ser deletados.

Aa estruturação de todos os elementos CSP que são capazes de reconhecer os casos de teste abstratos em CSP são agrupados, contendo todas as definições de tipos de dados, tuplas e canais usados para traduzir frases escritas em linguagem natural de alto nível em eventos CSP_M [Sca98].

Em [Lei06, Tor06] é possível obter uma melhor explicação sobre a definição de eventos CSP, usando tuplas, tipos de dados e canais, além de como esses eventos foram construídos tomando como base frases em inglês,

4.1.2.2 Geração de Modelos Formais Abstratos

O módulo de geração é responsável por criar modelos abstratos a partir de um conjunto de eventos CSP recebidos dos casos de teste. Para cada caso de teste é gerado um modelo correspondente. Um modelo abstrato gerado a partir de um único caso de teste é também chamado de caso de teste abstrato (ou formal). Casos de teste abstratos são representados tanto usando os formalismos CSP quanto LTS. Ao se utilizar o formalismo CSP, um modelo abstrato é visto como um processo CSP finito que termina em `SKIP`. Todos os eventos no processo CSP ocorrem sequencialmente usando o operador de prefixo (`->`). A Figura 4.5 mostra um caso de teste abstrato, em notação CSP, gerado a partir do script de teste TAF apresentado na Figura 4.4. Note que o conteúdo da Figura 4.5 é um processo CSP simples que foi construído basicamente a partir dos textos em Inglês contidos nos métodos description da Figura 4.4 traduzidos para CSP.

Na Figura 4.5 os eventos CSP **setup**, **test**, e **cleanup** foram colocados somente para analisar os eventos CSP gerados a partir de cada parte do caso de teste. Esses eventos não são gerados pelo do passo de processamento de linguagem natural.

O propósito da geração de casos de testes abstratos em CSP é a possibilidade de comparação entre cada caso de teste e o modelo comportamental do telefone celular a fim de encontrar uma possível implementação para cada evento *prototype* encontrado. Na Seção 4.3 é apresentado um algoritmo com essa finalidade.

```
TestCase TC_VIEW_AND_HIGHLIGHT_IN_OUTBOX_FLD:

 Setup:

  description("Delete all messages.");
   phone.goToIdle();
   phone.startApp(MESSAGING);
   phone.delete(ALL_MESSAGES);

  description("Set the phone to show the messages in the outbox folder by Subject.");
   phone.goToIdle();
   phone.startApp(MESSAGING);
   phone.goToAndSelectMenuItem(MESSAGE_SETUP);
   phone.scrollToAndSelect(FOLDER_VIEW);
   phone.scrollToAndSelect(SUBJECT);
   phone.waitForTransientScreen(Notice.CHANGED_FOLDER_VIEW_SUBJECT);

  description("Compose and send two messages to a valid phone number.");
   ...

  description("Go to the Outbox folder and verify that this screen is displayed.");
   ...

 Steps:

  description("Highlight the first message.");
   phone.scrollToMessage(Content.SMS_ONE);
   phone.verifyHighlight(Content.SMS_ONE);

  description("Highlight the second message.");
   phone.scrollToMessage(Content.SMS_TWO);
   phone.verifyHighlight(Content.SMS_TWO);

  description("Go to the Message Center and verify that this screen is displayed.");
   phone.goToAndSelectMenuItem(BACK);
   phone.checkScreen(MESSAGE_CENTER);

 PostCondition:

  description("Delete all messages.");
   ...

  description("Set the phone to show the messages in the folder by Address.");
   ...
```

Figura 4.4 Uma instância de um caso de teste automático (script de teste).

```
TC_VIEW_AND_HIGHLIGHT_IN_OUTBOX =

 setup ->
   delete.DTDEL_ITEM.(MESSAGES, {ALL}) ->
   set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_SUBJECT, {}) ->
   compose.DTWRITE_ITEM.(MESSAGES, {TWO}) ->
   send.DTSEND_ITEM.(MESSAGES, {TWO}).(NUMBER, {VALID}) ->
   goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}) ->
   verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}) ->

 test ->
   highlight.DTSELECT_ITEM.(MESSAGE, {FIRST}) ->
   highlight.DTSELECT_ITEM.(MESSAGE, {SECOND}) ->
   goTo.DTGOTO_ITEM.(OUTBOX_FOLDER, {}) ->
   verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {}) ->

 cleanup ->
   delete.DTDEL_ITEM.(MESSAGES, {ALL}) ->
   set.DTSET_ITEM.(VIEW_MESSAGES, {}).(BY_ADDRESS, {}) ->

 SKIP
```

Figura 4.5 Exemplo de caso de teste abstrato (em notação CSP).

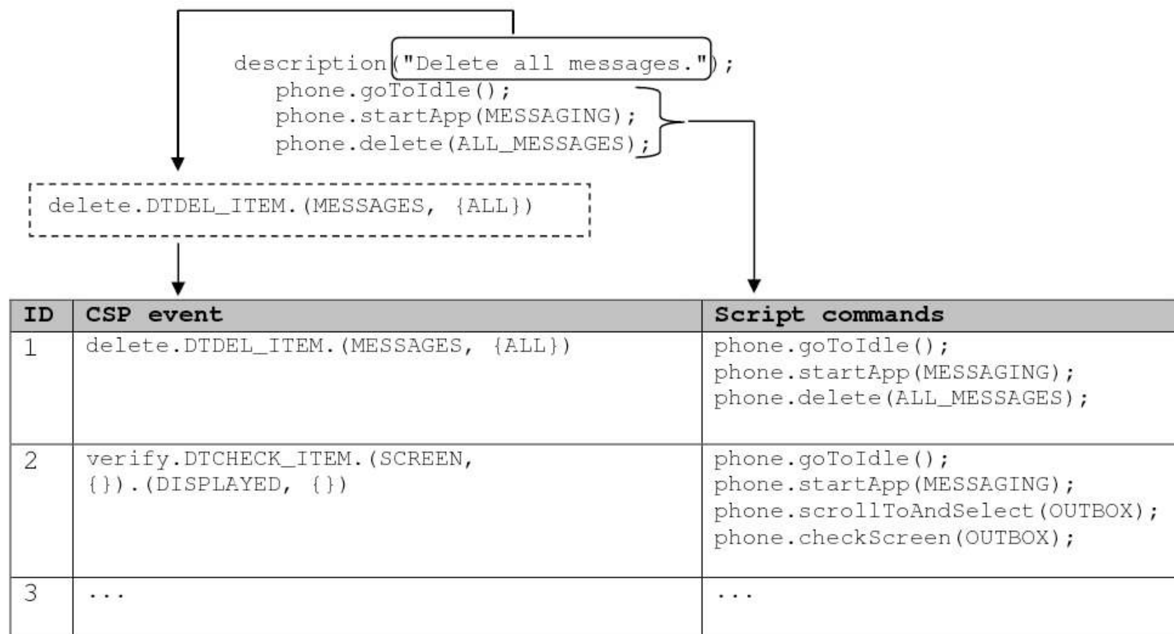
4.1.2.3 Associação de Eventos CSP com Comandos de Script

O módulo de mapeamento associa a cada evento CSP, seus comandos de script TAF correspondentes. O objetivo do mapeamento de eventos CSP em comandos de script é a possibilidade de capturar um caso de teste abstrato (em CSP) bem como um script de teste concreto (real). A Figura 4.6 mostra como é realizado o processo de mapeamento. Primeiro, o conteúdo do comando *description* (frase em inglês) é traduzido em um evento CSP. Depois de traduzido em um evento CSP, o conjunto de comandos de script correspondente é capturado para compor um novo mapeamento com o evento CSP. Esse novo mapeamento é armazenado como um novo dado de entrada em uma tabela de mapeamento. Uma tabela de um banco de dados relacional foi utilizada para armazenar todos os mapeamentos gerados.

Baseado na tabela de mapeamentos, é possível gerar novos scripts de teste através da busca nos dados já armazenados. A tabela de mapeamento é a base da estratégia de geração de novos scripts de teste onde os pares relacionados (CSP, TAF) possam ser reusados. O passo de geração segue uma tradução de casos de teste abstratos em scripts de teste concretos. Assim, para que tal geração seja realizada é necessário que exista a tabela de mapeamentos e um caso de teste abstrato em CSP. Mais informações sobre a geração de scripts de teste podem ser encontradas em [dS07].

4.1.3 Tradução de Casos de Teste Abstratos em Scripts de Teste

O passo de tradução de um caso de teste abstrato em um script de teste usa a tabela de mapeamentos gerada no passo de modelagem (Figura 4.6). A idéia é pegar o caso de teste abstrato e extrair todos os eventos CSP correspondentes. Para o passo de extração nós usamos um *parser* que captura todos os eventos CSP e os envia para o mecanismo de busca. Com todos os eventos



ID	CSP event	Script commands

1	delete.DTDEL_ITEM.(MESSAGES, {ALL})	phone.goToIdle();
		phone.startApp(MESSAGING);
		phone.delete(ALL_MESSAGES);

2	verify.DTCHECK_ITEM.(SCREEN, {}).(DISPLAYED, {})	phone.goToIdle();
		phone.startApp(MESSAGING);
		phone.scrollToAndSelect(OUTBOX);
		phone.checkScreen(OUTBOX);

3	...	...

Figura 4.6 Mapeamento de um evento CSP em um comando de script.

CSP extraídos do caso de teste abstrato, nós buscamos na tabela de mapeamento por todos os comandos de script correspondentes. Após extrair todos os comandos associados a cada evento CSP dado, nós organizamos os comandos em um arquivo separado. O resultado é um caso de teste automático como mostrado na Figura 4.4.

É importante salientar que existem situações onde nem todos os eventos CSP estão disponíveis (mapeados) na tabela de mapeamento. Nesses casos são gerados scripts de testes com elementos especiais. Esses elementos são identificados em um caso de teste pela tag ***prototype***. A Figura 4.7 mostra um trecho de um caso de teste gerado com um evento CSP não encontrado na tabela de mapeamentos.

```
prototype("PLAY.(MULTIMEDIA_FILE, {ONE})");
  /*  (“prototype” tag)
   *  This code is intended to be empty.
   *  A new code implementation must be put here.
   */
```

Figura 4.7 Exemplo da tag *prototype*.

Como mostrado na Figura 4.7 o evento CSP não encontrado
`PLAY.(MULTIMEDIA_FILE, {ONE})`
na tabela de mapeamento, é adicionado no script de teste uma tag de prototipação (**prototype**). A sequência de comandos são comentadas e deveriam ser codificadas manualmente após a geração. O presente trabalho tem como principal objetivo resolver essa limitação do método proposto por Souza em [dS07]. Para cada evento CSP não encontrado na tabela de mapeamentos, o próprio caso de teste abstrato é modificado, sendo adicionado um evento especial

também chamado *prototype*. Esse caso de teste abstrato atualizado em conjunto com o modelo comportamental do telefone descrito na Seção 4.2 são as duas principais entradas para a geração de código para as UFs do TAF que futuramente irão representar comandos de script para composição de casos de teste.

4.2 Extração Automática do Modelo Comportamental

O primeiro passo necessário para a geração automática de código para o TAF consiste da extração do modelo comportamental do telefone celular em que os testes automáticos estão sendo desenvolvidos. Para o presente trabalho, esse modelo é representado em CSP, com o objetivo de fazer uso das funcionalidades já disponíveis nesse formalismo para modelagem de sistemas, além do reuso de ferramentas já existentes provenientes de outros trabalhos do grupo de pesquisas do BTC.

Para obter o modelo de um telefone celular, nós implementamos uma ferramenta que denominamos BxT – *Behavior Extractor Tool* [NB07]. Através da execução dessa ferramenta, que requer um telefone celular plugado em uma porta USB do computador que irá extrair o modelo, é possível obter todos os caminhos relevantes da aplicação de um telefone celular Motorola através da análise do código-fonte de um *framework* de automação de testes proprietário da Motorola conhecido como PTF (*Phone Test Framework*) [EV06]. Além da geração completa, é possível produzir um modelo direcionado a um objetivo, ou seja, a BxT foi projetada para permitir ao usuário guiar a geração do modelo. Para realizar esse direcionamento basta o usuário fornecer como entrada um objetivo (por exemplo, uma aplicação ou um conjunto de textos presentes em um menu) e o modelo é gerado em torno desse objetivo. O princípio por atrás desse direcionamento para a geração do modelo é similar à abordagem utilizada em [CGP99] que propõe a geração de um modelo parcial, uma vez que um modelo completo nem sempre precisa ser gerado para que seja realizada a verificação de modelos (*model checking*). É importante salientar que a BxT extrai um comportamento de alto-nível de abstração do código-fonte, ou seja, um comportamento funcional relacionado principalmente a aspectos navegacionais. Esta é a principal razão pela qual os modelos formais que são extraídos a partir da implementação são bastante próximos dos modelos formais dos casos de teste (que por sua vez também são escritos em alto-nível de abstração). A seguir será detalhada como a extração do comportamento dos celulares é realizada.

Dada uma aplicação do telefone celular, BxT assume que o estado inicial é *Idle* (a tela inicial do telefone) e, usando a estrutura de chamadas de métodos do PTF, extrai todos os caminhos da aplicação especificada. Os resultados produzidos como saída após a execução da BxT são o modelo formal (descrito em CSP) bem como o log da chamada de métodos do PTF.

A Figura 4.8 mostra o engenho da BxT: para cada tela do telefone a ferramenta captura todos os componentes correspondentes aos itens retornados pelo PTF. Esses componentes podem ser telas, botões, labels, figuras, painéis, etc. Para cada componente, o PTF retorna um conjunto de propriedades, tais como texto, estado, cor, posição, etc. A BxT captura também os componentes de navegação, como por exemplo, o botão que foi pressionado para que uma determinada tela seja alcançada.

Um aspecto importante nessa geração é o reuso do módulo de processamento de lingua-

gem natural controlada (LNC) [CS06, dFLC06, Lei06] da ferramenta TaRGeT. O objetivo de se utilizar esse módulo é a obtenção de um alfabeto para a implementação que é compatível com o alfabeto obtido através da geração de casos de testes pela ferramenta TaRGeT. Nós simplesmente precisamos escrever a possível navegação encontrada pela BxT como uma frase em LNC (considerando o alfabeto da TaRGeT).

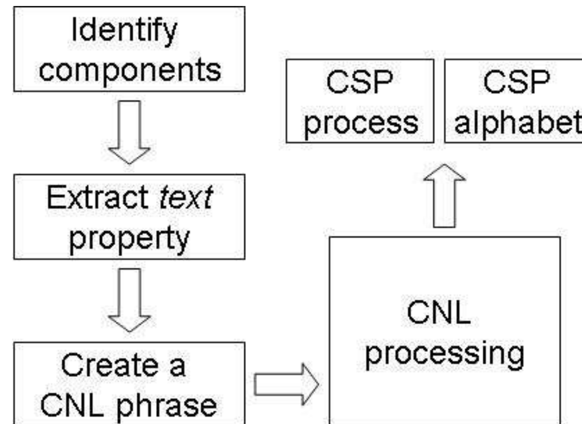


Figura 4.8 Engenho da ferramenta BxT.

Por exemplo, a navegação correspondente à seleção da aplicação de Mensagens no menu principal do telefone é expressa em LNC como "*Select Message*". Uma analogia pode ser vista graficamente na Figura 4.9. Através dessa figura, podem ser verificadas duas transições: da Tela (1) (*Idle*) para a Tela (2) (*Main Menu*), esta transição corresponde à parte "*Select*" da frase em LNC, e da Tela (2) para a Tela (3) (*Message Center*), que corresponde à parte "*Message*" da frase.

O resultado do uso da BxT em um telefone contendo uma aplicação similar à apresentada na Figura 4.9 é a especificação CSP ilustrada na Figura 4.10. Para a obtenção dessa especificação, utilizamos como objetivo a aplicação de IM (*Instant Messaging*). Para facilitar o entendimento da especificação CSP apresentada na Figura 4.10, nós criamos uma representação LTS dessa especificação que pode ser vista na Figura 4.12.

A vantagem dessa abordagem é a geração automática de um modelo formal. O usuário não precisa perder seu tempo para criar um modelo da aplicação e também não precisa ter expertise em métodos formais para saber como criar tal modelo. Nós observamos que algumas equipes de Testes da Motorola usam diagramas gráficos com vários propósitos e a geração desses diagramas é totalmente manual.

As Figuras 4.11 e 4.12 correspondem aos modelos de um caso de teste (`TC`) e de uma implementação (`System`) da aplicação relacionada ao caso de teste descritos em LTS, respectivamente. Através da observação desses modelos, podemos verificar que o LTS apresentado na Figura 4.12 praticamente contém o LTS apresentado na Figura 4.11. A diferença entre esses dois modelos aparecem basicamente por dois motivos: eventos de navegação em baixo-nível (*Go to main menu*, por exemplo) e transições extras não contempladas nos requisitos (que são fonte de entrada para geração de casos de teste). O primeiro motivo é natural e está relacionado com a implementação que é o artefato utilizado para originar o modelo. Já o segundo pode

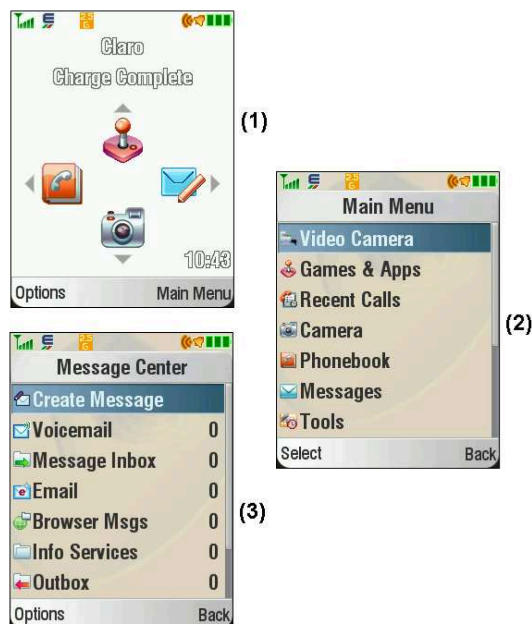


Figura 4.9 Exemplo de telas de um telefone celular.

ser considerado um problema, pois tais eventos expressam comportamentos não permitidos e assim são considerados erros de implementação. O presente trabalho não considera tal possibilidade, ou seja, para nós, a implementação que será utilizada para geração do modelo formal é sempre considerada correta.

É importante salientar que essas informações de baixo nível adquiridas através da execução da BxT são de fundamental importância para esse trabalho, pois essas informações em baixo nível representam as implementações das UFs. Entretanto, como dito previamente, a especificação originada através da implementação é mais rica em detalhes quando comparada às especificações dos casos de teste. Felizmente, como a parte do código-fonte necessário para originar o modelo é uma visão caixa-preta do telefone celular, o nível de abstração é muito próximo do modelo que representam os requisitos de onde os casos de teste são derivados.

Considerando as observações prévias sobre as Figuras 4.11 e 4.12, podemos constatar que para fazer uma comparação correta entre um caso de teste (`TC`) e a implementação (`System`) é necessário aplicar a seguinte relação de refinamento:

$$TC \sqsubseteq_T System \setminus Impl_details$$

onde $Impl_details = \alpha System \setminus \alpha TC$

Ou seja, para comparar a especificação `TC` e `System` corretamente é necessário esconder (usando o operador *hiding* de CSP) todos os eventos de `System` que não ocorrem em `TC`. Note que esses eventos dizem respeito aos detalhes adicionais de navegação contidos na implementação (`System`) quando comparados aos casos de teste (`TC`). Essa operação de *hiding* sobre o modelo gerado só deve ocorrer caso o evento especial *prototype* do `TC` não esteja associado ao evento que está sendo escondido. Assim, deve-se sempre verificar, antes de esconder um evento do modelo, se aquele evento não corresponde a um evento *prototype* to caso de teste.

```
System = P1_Hiding ; System

P1_Hiding = P1 \ {| goto.DTGOT_SCREEN.(IDLE_SCREEN, {}),
goto.DTGOT_MENU.(MAIN_MENU, {}) |}

P1 = steps -> goto.DTGOT_SCREEN.(IDLE_SCREEN, {}) ->
goto.DTGOT_MENU.(MAIN_MENU, {}) ->
start.DTSTA_APPLICATION.(IM_APPLICATION, {}) -> ( P2 [] PA1 )

P2 = steps -> login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {}) ->
(P3 [] SKIP)

P3 = steps -> select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN}) -> P4

P4 = steps -> start.DTSTA_LISTITEM.(CONVERSATION_ITEM, {}) -> P5

P5 = steps -> exit.DTEXI_APPLICATION.(IM_APPLICATION, {}) -> SKIP

PA1 = steps -> goto.DTGOT_LIST_ITEM.(SAVED_CONVERSATION_FOLDER, {}) ->
SKIP
```

Figura 4.10 Especificação CSP obtida através da execução da BxT em um telefone celular cujo objetivo era iniciar uma conversação e em seguida sair da aplicação de IM.

Considerando essa operação de abstração, os modelos (caso de teste e implementação) estão prontos para realizar a checagem através de refinamento e obter os contra-exemplos necessários para desenvolver as novas UFs necessárias para compor o TAF, fazendo com que o mesmo esteja preparado para executar o caso de teste em questão.

4.3 Obtenção de Representação CSP das UFs a Serem Desenvolvidas

Uma vez que já vimos como são obtidos os modelos `M` (correspondente ao comportamento extraído automaticamente do telefone) e `TC` (que corresponde ao caso de teste abstrato a ser automatizado), nós podemos então mostrar como as funcionalidades ainda não implementadas no TAF contidas em `TC` são resolvidas utilizando `M` como base.

Para ilustrar o funcionamento do método proposto, utilizaremos um exemplo hipotético cujos processos `TC` e `M`, que correspondem respectivamente às especificações referentes ao caso de teste e ao modelo comportamental do telefone, podem ser vistos na Figura 4.13. Note que existem dois eventos em `TC` denominados `p.1` e `p.2` (simplificações de *prototype*("...1"); e *prototype*("...2"); , respectivamente). Usamos esse evento para representar os eventos especiais *prototype* nesse script de teste. Nesse exemplo estamos considerando que a implementação da UF, representada por `p.1` e `p.2` , não existem para o telefone celular em questão (sob teste).

Especificamente para esse caso de teste hipotético, os eventos *prototype* (denotados por `p.1` e `p.2`) são os pontos onde devemos nos concentrar. Vamos explicar mais detalhadamente como obter `p.1` e depois, mais resumidamente como obter `p.2`. Considerando que temos o

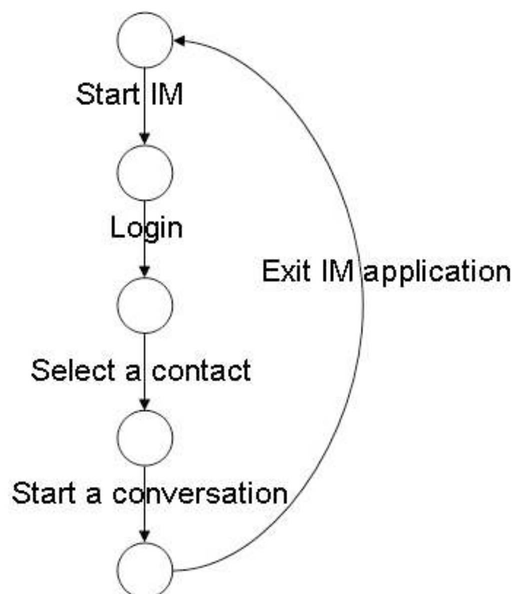


Figura 4.11 Representação de um caso de teste em LTS.

modelo extraído do telefone `M`, que pode ser visto na Figura 4.13, o nosso objetivo é encontrar em `M` o evento (ou os eventos) que representam o evento *prototype*.

Um simples refinamento não resolve o problema em questão, pois devemos considerar que um *prototype* pode ser representado por uma sequência de eventos. Quando uma relação de refinamento falha, somente um contra-exemplo (que representa o evento do modelo onde a falha pode ser observada) é retornado e com isso não conseguiríamos encontrar a solução do problema, caso o *prototype* em questão seja uma abstração de mais de um evento do modelo. Além disso, um único `TC` pode ter vários eventos *prototype* a serem resolvidos (como no exemplo acima). Para que possamos resolver todos eles, precisamos atualizar dinamicamente o próprio `TC` com eventos vindos de `M` (enquanto resolvemos cada *prototype*) para continuar com o refinamento até que o processo de terminação (`SKIP`) seja encontrado em `TC`. Além disso, também existe a necessidade de manter um mapeamento de todos os eventos *prototype* com os respectivos eventos de `M` que os representam.

A seguir apresentamos um algoritmo para lidar com os problemas acima, o qual é uma das principais contribuições desse trabalho. Como mencionado anteriormente, a pré-condição para a execução desse algoritmo é a obtenção dos modelos `M` e `TC` especificados em CSP. Para facilitar o entendimento inicial do algoritmo, iremos utilizar os processos apresentados na Figura 4.13.

Na primeira linha do algoritmo, definimos que o nosso mapeamento inicialmente, entre elementos *prototype* e eventos CSP, é uma sequência vazia. Cada elemento dessa sequência que representa o mapeamento é definido pela tupla

```
(p, <ev_1, ev_2, ..., ev_n>)
```

em que `p` representa um *prototype* contido em `TC` e `<ev_1, ev_2, ..., ev_n>` representa a sequência de eventos presentes em `M` que devem substituir `p` em `TC` a fim de validar

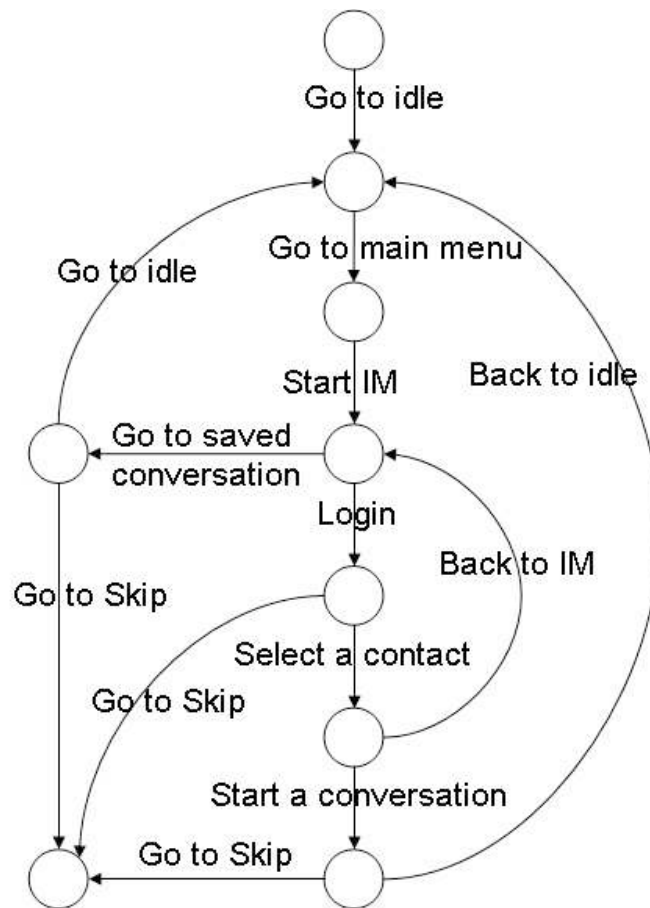


Figura 4.12 Representação de uma implementação em LTS.

o TC de acordo com o modelo M. Assim, um mapeamento de dois eventos *prototype* (p.x e p.y), cujos eventos correspondentes são a, b e c, respectivamente, deve ser representado da seguinte forma:

```
<(p.x, <a, b>), (p.y, <c>)>.
```

A segunda linha do algoritmo atribui a tr o trace resultante do refinamento TC_M $\sqsubseteq_F$ TC, em que TC_M é definido como o paralelismo generalizado entre os processos TC e M

```
TC_M = TC [| Events |] M
```

e Events representa todos os eventos do alfabeto.

A linha 3 atribui a pt o último evento do trace tr. Para isso precisamos definir uma função denominada *last*(), que dado um trace tr, retorna o último elemento encontrado no mesmo. A Função 4.1 apresenta a definição dessa função.

$$last(tr\hat{\ }\langle e\rangle) = e \tag{4.1}$$

```
TC = a -> b -> p.1 -> e -> f -> p.2 -> h -> SKIP

M = a -> (d -> a -> j -> M
         [] b -> c -> d -> e -> f -> (g -> h -> SKIP
                                      [] k -> l -> M)).
```

Figura 4.13 Exemplo de processos CSP (caso de teste abstrato e modelo comportamental).

Exemplo 4.1 *Seja tr=*$\langle a,b,c \rangle$ *um trace. Então aplicando a função last(.) sobre tr, obtemos last(tr)=c.*

Após atribuir os respectivos valores a `tr` e `pt`, temos a seguir um laço que tem duas condições de parada: a primeira verifica se não há mais problemas a serem resolvidos, ou seja, se depois de substituir todos os eventos *prototype* pelos seus respectivos valores, o modelo `M` refina `TC`; a segunda verifica se o último evento de `tr` é um ✓ (`tick`). Esse evento indica que o modelo `M` refina o `TC` e os dois chegaram ao fim em um evento que representa uma terminação com sucesso (`SKIP`). Considerando que temos como pré-condição que `TC` seja um caso de teste abstrato com pelo menos um problema (*prototype*) a ser resolvido, o algoritmo executa o código contido no escopo desse laço pelo menos uma vez.

Algoritmo 1 Algoritmo Proposto

1: $mapping \leftarrow \langle\rangle$
2: $tr \leftarrow TC_M \sqsubseteq_F TC$
3: $pt \leftarrow last(tr)$
4: **while** $(tr \neq \langle\rangle$ **and** $pt \neq \checkmark)$ **do**
5: $\quad deep \leftarrow \#tr$
6: $\quad event \leftarrow last(FindTr(deep) \sqsubseteq_F M \,|[Events]|\, Build(front(tr)))$
7: $\quad nextTCEvent \leftarrow getNextEvent(pt,\ TC)$
8: $\quad$ **while** $(event \neq nextTCEvent)$ **do**
9: $\qquad updateMapping(mapping,\ pt,\ event)$
10: $\qquad i \leftarrow deep\ +\ \#getMapping(mapping,\ pt)$
11: $\qquad event \leftarrow last(FindTr(i) \sqsubseteq_F M \,|[Events]|\, Build(front(tr) \frown getMapping(mapping, pt)))$
12: $\quad$ **end while**
13: $\quad updateTC(TC, pt,\ getMapping(mapping,\ pt))$
14: $\quad tr \leftarrow TC_M \sqsubseteq_F TC$
15: $\quad pt \leftarrow last(tr)$
16: **end while**

A primeira linha do laço (linha 5 do Algoritmo 1) atribui a `deep` um valor inteiro que indica o tamanho atual do trace correspondente a `tr`. Considerando os modelos apresentados na Figura 4.13, o valor de `tr` inicialmente é `<a,b,p.1>`, que representa o primeiro contra-

exemplo retornado por $\texttt{TC_M} \sqsubseteq_F \texttt{TC}$, e `deep`, por sua vez, recebe o valor 3 que indica o número de elementos de `tr`. O valor de `deep` será utilizado a seguir pelo processo *FindTr*() que dispõe de todos os eventos do alfabeto até o tamanho desejado para ser usado na descoberta do evento correspondente ao evento `p.1`. Esse processo é definido como

$$\begin{aligned} &FindTr(0) = Skip \\ &FindTr(deep) = |\sim| \ ev : Events \ @ \ ev -> FindTr(deep-1) \end{aligned} \tag{4.2}$$

Exemplo 4.2 *Seja deep = 3 um inteiro e* $\Sigma = \{a,b,c,d,e\}$ *o alfabeto da especificação. Então o processo FindTr(.) quando aplicado sobre deep retorna FindTr(deep)=*$a \rightarrow b \rightarrow c \rightarrow Skip$.

Basicamente o processo *FindTr*() fornece um comportamento que possibilita encontrar o evento de `M` que substitui parcial ou completamente os eventos *prototype* de `TC` (representado por `p.1` e `p.2` na Figura 4.13). Outro processo necessário para que obtivéssemos os resultados esperados é descrito a seguir.

$$\begin{aligned} &Build(\langle\rangle) = [\,]\ ev : Events \ @ \ ev -> Skip \\ &Build(\langle ev\rangle\hat{\ }t) = ev -> Build(t) \end{aligned} \tag{4.3}$$

Exemplo 4.3 *Seja tr =* $\langle a,b\rangle$ *um trace. Então o processo Build(.) quando aplicado sobre tr retorna Build(tr)=*$a \rightarrow b \rightarrow Skip$.

Build() recebe como parâmetro uma sequência e cria um processo composto pelos elementos dessa sequência. Assim como *FindTr*(), o processo *Build*() também finaliza em um processo de terminação com sucesso. Note que a linha número 6 do algoritmo proposto utiliza tanto *FindTr*() quanto *Build*(). Nesse ponto, o algoritmo executa o refinamento

$$FindTr(deep) \sqsubseteq_F M|[Events]|Build(front(tr))$$

que retorna `<a,b,c>` na primeira vez que é executado. Aqui definimos uma função denominada *front*() que é apresentada na Função 4.4. Essa função recebe um trace como parâmetro e retorna o mesmo trace, excluindo dele o último evento. Considerando o nosso exemplo, o trace `<a,b>` é retornada pela função *front*(`tr`) (observe que o evento `p.1` é removido do trace). O trace resultante desse refinamento é enviado para a função *last*(), que por sua vez retorna o evento `c` como resultado. Esse valor é atribuído a `event` e será adicionado ao mapeamento do *prototype* `p.1`.

$$front(tr\hat{\ }\langle e\rangle) = tr \tag{4.4}$$

Exemplo 4.4 *Seja tr=*$\langle a,b,c\rangle$ *um trace. Então aplicando a função front(.) sobre tr, obtemos front(tr)=*$\langle a,b\rangle$.

A próxima linha do algoritmo executa a função *getNextEvent*() (Função 4.5) que recebe dois parâmetros: o primeiro é um evento (correspondente ao *prototype* que nesse momento no nosso exemplo será `p.1`) e o trace `TC`. Essa função retorna o evento de `TC` que ocorre imediatamente após `p.1`. Considerando o `TC` definido na Figura 4.13, *getNextEvent*() irá retornar o evento `e`. A motivação para capturar este evento seguinte é permitir que nosso algoritmo saiba onde o *prototype* potencialmente termina (nesse caso no evento `e`).

$$
\begin{array}{l}
getNextEvent(ev,\ \langle\rangle) = ev \\
getNextEvent(ev, tr)\ = \\
\quad if\ head(tr)\ ==\ ev\ then \\
\qquad head(tail(tr)) \\
\quad else \\
\qquad getNextEvent(ev,\ tail(tr))
\end{array}
\tag{4.5}
$$

Exemplo 4.5 *Seja p=b um evento e tc=$\langle a,b,c,d\rangle$ um trace. Então aplicando a função getNextEvent(.,.) sobre p e tc, obtemos getNextEvent(p,tc) = c.*

Entramos então no laço mais interno que objetiva resolver completamente um *prototype*, ou seja, verificar se o `event` obtido já satisfaz ou se é necessário buscar mais eventos em `M` para substituir `p.1` em `TC`. Dentro do laço mais interno, a estrutura `mapping` é atualizada pela função *updateMapping*() (Função 4.6). São passados como parâmetro para essa função o mapeamento `mapping`, o evento correspondente ao *prototype* em questão (`pt`) e o evento que irá compor a sequência de eventos para o *prototype* vindo de `M`. Essa função verifica se o *prototype* já está na estrutura `mapping`, através da chamada da função *getMapping*() (Função 4.8) que será detalhada no próximo parágrafo. Caso a Função 4.8 retorne um trace não vazio, esse trace retornado (correspondente trace que mapeia o *prototype* em questão) é atualizado, sendo o evento passado como parâmetro adicionado ao final da sequência atual. Caso contrário, uma nova tupla é criada, baseada nos parâmetros informados à função, e adicionada ao mapeamento. A atualização de um mapeamento se dá através da remoção do mapeamento atual para o evento *prototype* em questão e da adição de um novo mapeamento atualizado. A remoção de um mapeamento é realizada pela função `removeMapping()` apresentada pela Função 4.7.

$$
\begin{array}{l}
updateMapping(mapping,\ ev1,\ ev2) = \\
let \\
\quad s = getMapping(mapping, ev1) \\
within \\
\quad if\ \#s == 0\ then \\
\qquad mapping\hat{\ }\langle(ev1, \langle ev2\rangle)\rangle \\
\quad else \\
\qquad removeMapping(mapping, ev1)\hat{\ }\langle(ev1, s\hat{\ }\langle ev2\rangle)\rangle
\end{array}
\tag{4.6}
$$

Exemplo 4.6 *Seja m=*$\langle(a,\langle b,c\rangle),(d,\langle g,f\rangle)\rangle$ *uma sequência, p=a e q=h dois eventos. Então aplicando a função updateMapping(.,.,.) sobre m, p e q, obtemos updateMapping(m,p,q) =* $\langle(d,\langle g,f\rangle),(a,\langle b,c,h\rangle)\rangle$.

$$removeMapping(mapping,ev) = \langle (ev1,ev2) \mid (ev1,ev2) \leftarrow mapping,\ ev1\ !=\ ev\rangle \qquad (4.7)$$

Exemplo 4.7 *Seja m=*$\langle(a,\langle b,c\rangle),(d,\langle g,f\rangle)\rangle$ *uma sequencia, p=a um evento. Então aplicando a função removeMapping(.,.) sobre m e p, obtemos removeMapping(m,p) =* $\langle(d,\langle g,f\rangle)\rangle$.

A linha 10 do algoritmo faz uma chamada à função *getMapping*() (Função 4.8) que recebe o mapeamento e o evento *prototype* cujo trace que o representa será retornado. Assim, à variável `i` é atribuído o valor de `deep` adicionado ao tamanho do trace que representa o *prototype* em questão.

$$\begin{array}{l} getMapping(\langle\rangle,event)\ =\ \langle\rangle \\ getMapping(mapping,event)\ = \\ \quad head(\langle ev \mid (m,ev)\ \leftarrow\ mapping\ m == event\rangle) \end{array} \qquad (4.8)$$

Exemplo 4.8 *Seja m=*$\langle(a,\langle b,c\rangle),(d,\langle g,f\rangle)\rangle$ *uma sequencia e p=d um evento. Então aplicando a função getMapping(.,.) sobre m e p , obtemos getNextEvent(m,p)=*$\langle g,f\rangle$.

A última linha do laço mais interno executa mais uma vez o refinamento que tem por objetivo descobrir o próximo evento que irá representar o *prototype*. Note que novos valores para os parâmetros são utilizados nesse ponto, quando comparado à sua última utilização (linha 6). Esses valores já consideram os valores contidos no mapeamento que está sendo composto e armazenado em `mapping`.

Ao finalizar a primeira execução do laço que inicia na linha 8, `mapping` armazena o valor `<(p.1, <c,d>)>`. Ao sair do laço mais interno, verifica-se a chamada da função *updateTC*() (Função 4.9) que recebe como parâmetro o `TC` que será atualizado, `pt` que representa o *prototype* que será atualizado e *getMapping*() que retorna o trace que representa o *prototype* que será substituído em `TC`. Ao executar a função *updateTC*() no exemplo acima, o resultado é a atualização do processo `TC` que tem `p.1` pelo seu comportamento correspondente `<c,d>`. Assim o novo `TC` é representado por

```
TC = a -> b -> c -> d -> e -> f -> p.2 -> h -> SKIP
```

Note que após realizar essa atualização, `TC` ainda não é um trace válido do processo `M`, pois ainda há um *prototype* (`p.2`) a ser resolvido. Observe também que a função *updateTC*() utiliza duas funções auxiliares para realizar o seu processamento. A função *lastTC*() (Função 4.10) recebe um evento e um trace como parâmetros e retorna um sub-trace a partir do evento que

foi passado como parâmetros, removendo assim os primeiros eventos. Já a função $frontTC()$ (Função 4.11) faz o contrário, ou seja, retorna o sub-trace que representa os eventos iniciais até encontrar o evento passado como parâmetro, removendo o final da lista. As duas funções juntas tem por objetivo remover o evento correspondente ao *prototype* a fim de que o mesmo seja substituído pelo trace que o representa (encontrado após a execução do laço mais interno do algoritmo).

$$updateTC(tc, pt, map) = frontTC(pt, tc)\hat{\ }map\hat{\ }lastTC(pt, tc) \tag{4.9}$$

Exemplo 4.9 *Seja tc=*$\langle a, b, p1, e\rangle$ *um trace, pt=*$p1$ *um evento e map=*$\langle(p1, \langle c, d\rangle)\rangle$ *uma sequência. Então aplicando a função updateTC(.,.,.) sobre tc, pt e map, obtemos updateTC(tc, pt, map)=*$\langle a, b, c, d, e\rangle$.

O algoritmo continua sua execução. Ao executar a linha 14, o valor de `tr` é atualizado, armazenando agora o trace `<a,b,c,d,e,f,p.2>`. A linha 15 do algoritmo dessa vez atrubui à variável `deep` o valor `p.2`. Observe que a condição do looping mais externo (linha 4) não é satisfeita, e então o algoritmo continua sua execução. `deep` passa a armazenar o valor 7, `event = g` e `nextTCEvent = h`. Passamos então a executar o laço mais interno da linha 8. A condição do while é satisfeita e a função $updateMapping()$ é executada, atualizando o conteúdo de `mapping` que passa agora a ser representado por `<(p.1, <c,d>), (p.2, <g>)>`. O laço mais interno encerra sua execução e então a função $updateTC()$ é executada e atualiza `TC` que passa a ser representado por

```
TC = a -> b -> c -> d -> e -> f -> g -> h -> SKIP
```

Note que após realizar essa atualização, `TC` passa a ser um trace válido do processo `M` e assim chegamos ao final da execução do Algoritmo 1.

É importante salientar que utilizamos o modelo de falhas para realizar os refinamentos porque precisamos obter os contra-exemplos das falhas que acontecem quando comparamos o modelo comportamental dos telefones `M` com `TC`.

$$\begin{aligned}
&lastTC(pt, \langle\rangle) = \langle\rangle \\
&lastTC(pt, \langle ev\rangle\hat{\ }t) = \\
&\quad \textit{if } ev! = pt \textit{ then} \\
&\quad\quad lastTC(pt, t) \\
&\quad \textit{else} \\
&\quad\quad t
\end{aligned} \tag{4.10}$$

Exemplo 4.10 *Seja pt=p1 um evento e tc=*$\langle a, b, p1, e\rangle$ *um trace. Então aplicando a função lastTC(.,.) sobre pt e tc, obtemos lastTC(pt,tc)=*$\langle e\rangle$.

$$
\begin{array}{l}
\mathit{frontTC}(pt, \langle\rangle) = \langle\rangle \\
\mathit{frontTC}(pt, \langle ev\rangle\hat{\ }t) = \\
\quad \mathit{if}\ ev! = pt\ \mathit{then} \\
\quad\quad \langle ev\rangle\hat{\ }\mathit{frontTC}(pt, t) \\
\quad \mathit{else} \\
\quad\quad \langle\rangle
\end{array}
\tag{4.11}
$$

Exemplo 4.11 *Seja pt=p1 um evento e tc=*$\langle a,b,p1,e\rangle$ *um trace. Então aplicando a função frontTC(.,.) sobre pt e tc, obtemos frontTC(pt,tc)=*$\langle a,b\rangle$.

Por fim, a geração do código que representa o trace `<c,d>` é obtida de forma simples. Lembre-se que mencionamos anteriormente que além do modelo comportamental a BxT também armazena todas ações realizadas no telefone celular. Assim, fazemos uma busca na estrutura de dados que armazena o código PTF correspondente à execução que gerou o trace `<c,d>` e criamos um novo arquivo com o código correspondente e o adicionamos à biblioteca de UFs do TAF. Após realizar esse último passo, a nova funcionalidade, já codificada em uma nova UF, pode ser utilizada por casos de testes TAF. No Capítulo 5 iremos apresentar um estudo de caso mostrando a aplicação do método proposto para resolver um problema real.

Antes de iniciar o próximo capítulo, iremos descrever algumas limitações que encontramos no algoritmo proposto após fazer uma análise detalhada do mesmo. A seguir essas limitações serão enumeradas.

1. Não faz sentido para o algoritmo existirem dois eventos *prototype* em sequência, como no exemplo

   ```
   TC = a -> b -> pt.1 -> pt.2 -> c -> SKIP
   ```

   Nesse caso é impossível para o algoritmo identificar onde `pt.1` acaba e `pt.2` começa.

2. Também não é possível identificar a possível implementação de um *prototype* caso este seja o evento que precede o processo de terminação com sucesso em `TC`, como mostra o exemplo a seguir

   ```
   TC = a -> b -> c -> pt.1 -> SKIP
   ```

   Considerando esse exemplo, não necessariamente a BxT produzirá um modelo `M` que termine no mesmo ponto (modelos tendem a ser recursivos) e assim o algoritmo muito provavelmente não produzirá uma saída válida.

3. O exemplo a seguir também mostra uma limitação do algoritmo proposto:

   ```
   TC = a -> pt.1 -> c -> SKIP
   M = a -> b -> c -> d -> c -> SKIP
   ```

Nesse caso o *prototype* `pt.1` deveria ser representado pelo trace `<b,c,d>`, porém o algoritmo irá retornar o trace `<b>` como resultado, pois ele utiliza o próximo evento de `TC` para comparação na condição de parada. Porém, considerando o domínio da aplicação (telefonia celular), a possibilidade desse problema acontecer é relativamente baixa.

4. O algoritmo proposto também não considera a possibilidade de obter múltiplas implementações, como no exemplo abaixo

```
TC = a -> b -> pt.1 -> d -> SKIP
M = a -> b -> (c -> d -> SKIP)
              []
              (e -> d -> SKIP)
              []
              (f -> d -> SKIP)
```

   Observe que no exemplo acima o valor de `pt.1` poderia ser tanto `<c>` quanto `<e>` quanto `<f>`, porém o algoritmo que desenvolvemos considera somente o primeiro contra-exemplo retornado pelo FDR, ou seja, o algoritmo acha a menor implementação para a UF representada por `pt.1`.

5. Por fim, o modelo comportamental `M` precisa obrigatoriamente oferecer o próximo evento após o *prototype*, caso contrário o algoritmo falha (não encontra o próximo evento de `TC`). Pra esclarecer o problema, apresentamos o exemplo a seguir. Note que o evento `d` não é oferecido por `M`.

```
TC = a -> b -> pt.1 -> d -> SKIP
M = a -> b -> c -> e -> SKIP
```

   Porém, como dito anteriormente, assumimos que `M` é coerente e esse caso não ocorre.

CAPÍTULO 5

Estudo de Caso

Nesse capítulo apresentaremos um estudo de caso realizado na Motorola com o objetivo de avaliar a abordagem apresentada no Capítulo 4. Nesse estudo de caso, nós consideramos todo o processo para obtenção dos resultados, ou seja, iremos mostrar todo o fluxo desde o processamento do documento de requisitos até a geração do código de uma UF (*Utility Function*) para o TAF (*Test Automation Framework*) [KKR+07]. Para realizar esse estudo, utilizamos um documento de requisitos da *feature* de IM (*Instant Messaging*) escrito de acordo com as regras descritas em [dFLC06]. Nós escolhemos essa feature pelo fato dela descrever cenários interessantes para aplicação do método proposto, além de descrever vários cenários interessantes para automação de testes.

O cenário utilizado foi o da *funcionalidade de conversação* utilizando a aplicação de IM dos celulares Motorola. Essa aplicação tem por objetivo prover a comunicação do usuário com outros usuários do mesmo serviço que estão presentes em sua lista de contatos, podendo esses outros usuários estarem logados através dos softwares tradicionais de IM que rodam em sistemas operacionais de máquinas tradicionais ou em outros dispositivos móveis. Geralmente a lista de contatos fica armazenada no servidor e é importada para o telefone após realizado o login. A aplicação de IM presente nos celulares da Motorola é capaz de acessar os servidores que provêem esse tipo de serviço, desde que a operadora suporte e configure o serviço nos dispositivos. Atualmente esse serviço é largamente utilizado em países como Estados Unidos e em países da Europa, mas ainda não é utilizado em larga escala no Brasil.

A seguir iremos descrever uma visão geral da abordagem proposta e detalhar cada um dos passos necessários para a obtenção dos resultados apresentados.

5.1 Visão Geral da Abordagem

A Figura 5.1 apresenta uma visão geral dos passos necessários para a obtenção dos resultados apresentados nesse capítulo. É importante salientar que a abordagem aqui proposta visa a geração de código de UFs para o TAF de forma automática. A seguir iremos descrever cada um dos estados apresentados no fluxograma da Figura 5.1.

O primeiro fluxo mostra a ferramenta TaRGeT (*Test and Requirements Generation Tool*) [NCT+07] aceitando como entrada um conjunto de documentos de requisitos. Nesse primeiro momento, a ferramenta utiliza a abordagem de geração de modelos formais a partir de documentos de requisitos proposta em [dFLC06]. Esse trabalho define um processo para gerar um modelo formal a partir documentos de requisitos usuais, ou seja, esses documentos são escritos em inglês e seguem um template de caso de uso definido para facilitar a conversão *LNC ->*

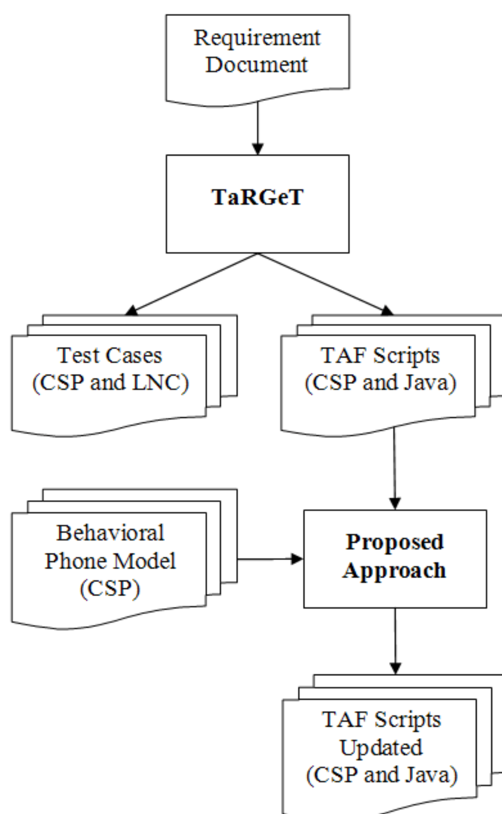


Figura 5.1 Visão geral da abordagem proposta.

CSP. A Figura 5.2 mostra como funciona esse processo de geração de modelos formais a partir de documentos de requisitos.

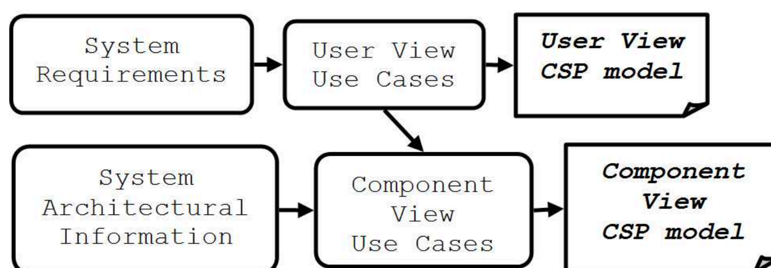


Figura 5.2 Fluxo do processo de geração de modelos formais a partir de documentos de requisitos.

De acordo com a Figura 5.2, dois modelos formais são gerados: *User View CSP Model* e *Component View CSP Model*. O primeiro modelo é gerado a partir de documentos de requisitos enquanto o segundo é gerado a partir de documentos de arquitetura. Inicialmente, ambos os documentos (de requisitos e de arquitetura) devem ser escritos utilizando respectivamente os templates *User View Use Cases* e *Component View Use Cases*. Depois deles terem sido escritos no padrão de caso de uso, é possível gerar modelos CSP automaticamente. É importante sali-

entar que somente o *user view model* é considerado para a obtenção de resultados no presente trabalho.

```
UC 01 - IM Conversation

Related requirement(s): REQ_1201, REQ_1218
Description: User starts a conversation in IM application.

Main Flow
From Step:  START
To Step:    END
```

Step Id	User Action	System State	System Response
1M	Start IM application.		IM application is started.
2M	Login.		Contact List is shown.
3M	Select a contact.		Conversation is opened.
4M	Exit IM application.		Phone goes to idle.

```
Exceptions Flow
From Step:  3M
To Step:    END
```

Step Id	User Action	System State	System Response
1E	Select an offline contact.	There at least one contact in offline contact list.	Dialog is shown asking if conversation should be opened.
2E	Confirm conversation dialog.		Conversation is opened.

Figura 5.3 Um exemplo de um *user view use case*.

A Figura 5.3 representa um *user view use case*. Sumarizando, um *user view use case* é dividido em dois fluxos: o fluxo principal e o de exceções. Para cada fluxo é definido o ponto inicial (*From Step:*) e o ponto final (*To Step:*) que definem o início e o fim do fluxo da execução. Cada passo do caso de uso contém os seguintes campos: *step identification* (que representa a identificação única de cada passo do caso de uso, *user action* (que representa a ação do usuário), *system state* (que representa o estado atual do sistema para que aquele passo possa ser executado) e *system response* (que indica a resposta do sistema àquela determinada ação).

A tradução consiste em pegar o conteúdo dos campos *User Action* e *System Response* de cada passo e enviá-los para o sistema de processamento de linguagem natural apresentado em [Lei06]. O resultado do processamento de cada um dos campos é um evento CSP do *user view model*.

O conteúdo do campo *System State* define se um fluxo de exceção deve ser seguido ou não. Os fluxos de exceção são introduzidos no modelo CSP utilizando escolhas externas (`[ ]`). Finalmente, o caso de uso é traduzido para um processo CSP. A Figura 5.4 mostra o processo

CSP gerado a partir do caso de uso apresentado na Figura 5.3.

```
UC_02_1M = (
 steps -> start.DTSTA_APPLICATION.(IM_APPLICATION, {}) ->
 expectedResults ->
    display.DT_VALUE.(SCREEN, {IM_APPLICATION}).(DISPLAYED, {}) ->
  UC_02_2M
)

UC_02_2M = (
 steps -> login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {}) ->
 expectedResults ->
    display.DT_VALUE.(SCREEN, {CONTACT_LIST}).(DISPLAYED, {}) ->
  UC_02_3M
 []
  UC_02_5E
)

UC_02_3M = (
 steps -> select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN}) ->
 expectedResults ->
    display.DT_VALUE.(SCREEN, {CONVERSATION_ITEM}).(DISPLAYED, {})) ->
  UC_02_4M
 []
  UC_02_1E
)

UC_02_4M = (
 steps -> exit.DTEXI_APPLICATION.(IM_APPLICATION, {}) ->
 expectedResults ->
    display.DT_VALUE.(SCREEN, {IDLE}).(DISPLAYED, {}) ->
  SKIP
)
```

Figura 5.4 Processo CSP gerado a partir do caso de uso apresentado na Figura 5.3.

Cada passo do caso de uso é traduzido em um sub-processo CSP (como *UC_02_1M* e *UC_02_2M*). O fluxo de exceções é introduzido por uma escolha externa entre os sub-processos principal e de exceção (por exemplo, *UC_02_3M* e *UC_02_5E*).

Nesse ponto, o segundo fluxo apresentado na Figura 5.1 inicia. Após traduzir os requisitos escritos em linguagem natural para CSP, a ferramenta TaRGeT continua o seu processamento a fim de gerar casos de teste de acordo com um critério de seleção denominado propósito de teste [dCN06]. Um propósito de teste, nesse caso, pode ser visto como um evento ou um trace presente na especificação CSP dos requisitos. Assim, todos os traces que satisfazem aquele determinado critério são selecionados e depois é feita uma triagem, restando somente os casos de teste realmente válidos para aquele critério de seleção. Cada um dos traces resultantes origina um novo caso de teste. Nesse momento, a ferramenta produz como saída um conjunto de casos de teste que satisfazem um determinado propósito de testes previamente definido. Esses casos de teste são representados tanto em linguagem natural [Tor06] quanto em CSP

[dCN06].

Ainda no segundo fluxo da Figura 5.1, entra em execução o trabalho apresentado em [dS07], que tem como um de seus objetivos a tradução de cada caso de teste gerado por [dCN06] em scripts de testes para o TAF. Assim como ocorre com a geração de casos de teste descrita anteriormente, nesse momento também são gerados scripts de teste TAF tanto na própria linguagem do TAF (Java) quanto em CSP. Uma vez que novos casos de teste estão sendo gerados, é provável que nem todos os comandos necessários para compor o script de teste estejam mapeados no banco de dados, como mostrado em [dS07]. Assim, se uma entrada supostamente não existir na tabela de mapeamentos, uma tag especial deve ser adicionada no script de teste (tanto no script TAF quanto em sua representação CSP). Essa tag especial, denominada *prototype*, é de fundamental importância para esse trabalho, pois ela indica no caso de teste o ponto exato onde uma nova UF precisa ser implementada. Um exemplo de um script de teste com essa tag especial é apresentado na Figura 5.5. Note que o passo 2, referente ao login do usuário na aplicação de IM é representado pelo comando especial `prototype("imTk.loginIM()")`. Mais adiante, na Figura 5.7, é apresentada uma representação desse mesmo caso de teste em CSP.

```
// Step 1.P: Start IM application
navigationTk.launchApp(PhoneApplication.IM);

// Step 1.ER: IM application is started
phoneTk.checkScreen(PhoneScreen.IM);

// Step 2.P: Login
prototype("imTk.loginIM()");

// Step 2.ER: Contact List is shown
phoneTk.checkScreen(IMScreen.CONTACT_LIST);

// Step 3.P: Select a contact
imTk.scrollToContact(IMContact.JOHN);

// Step 3.ER: Conversation is opened
phoneTk.checkScreen(IMScreen.CONVERSATION);

// Step 4.P: Exit IM application
imTk.exitIM();

// Step 4.ER: Phone goes to idle
phoneTk.checkScreen(PhoneScreen.IDLE);
```

Figura 5.5 Fragmento de um script de teste prototipado em TAF.

A representação CSP do script de teste automatizado é a primeira entrada para o nosso método. A outra entrada é obtida através da execução da ferramenta BxT (*Behavior Extractor Tool*), apresentada na Seção 4.2 do Capítulo 4. A BxT deve ser executada no telefone celular cujo caso de teste prototipado deve ser criado. Não necessariamente todos os telefones (de famílias diferentes) irão gerar modelos comportamentais diferentes, mas não é possível prever isso. Dessa forma, é gerado o modelo comportamental para cada família de telefones em

que o teste se aplica a fim de que possamos gerar uma nova implementação de UF baseada no modelo CSP (como o modelo apresentado na Figura 5.6), que representa o comportamento de cada uma dessas famílias. A Figura 5.6 apresenta um exemplo de especificação CSP obtida através da execução da BxT em um telefone celular cujo objetivo era iniciar uma conversação e em seguida sair da aplicação de IM. Após integradas no TAF, através de investigação humana, é possível unificar as implementações geradas para todos os telefones das famílias que apresentem o mesmo comportamento, porém o presente trabalho não considera tal investigação. O presente trabalho apenas considera a geração de novas UFs, tendo em vista que as mesmas não existem até então.

```
System = P1_Hiding ; System

P1_Hiding = P1 \ {| goto.DTGOT_SCREEN.(IDLE_SCREEN, {}),
goto.DTGOT_MENU.(MAIN_MENU, {}) |}

P1 = steps -> goto.DTGOT_SCREEN.(IDLE_SCREEN, {}) ->
goto.DTGOT_MENU.(MAIN_MENU, {}) ->
start.DTSTA_APPLICATION.(IM_APPLICATION, {}) -> ( P2 [] PA1 )

P2 = steps -> login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {}) ->
(P3 [] SKIP)

P3 = steps -> select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN}) -> P4

P4 = steps -> start.DTSTA_LISTITEM.(CONVERSATION_ITEM, {}) -> P5

P5 = steps -> exit.DTEXI_APPLICATION.(IM_APPLICATION, {}) -> SKIP

PA1 = steps -> goto.DTGOT_LIST_ITEM.(SAVED_CONVERSATION_FOLDER, {}) ->
SKIP
```

Figura 5.6 Exemplo de especificação CSP obtida através da execução da BxT.

Para que fosse possível gerar código a partir do modelo comportamental obtido através da execução da ferramenta BxT, nós projetamos essa ferramenta para armazenar, além do modelo comportamental em CSP, informações referentes à navegação e às telas do telefone. Assim, para um determinado evento em CSP, temos uma estrutura de dados a parte onde armazenamos quais teclas foram pressionadas para chegar em determinado ponto, bem como as informações existentes em determinada tela.

Até aqui mostramos como ocorre todo o fluxo necessário para a obtenção das entradas requeridas pelo método proposto, como pode ser visto na Figura 5.1. A seguir iremos detalhar a aplicação do método proposto, considerando as duas entradas obtidas.

5.2 Aplicação do Método Proposto

Para ilustrar o funcionamento do método proposto, utilizaremos o caso de teste apresentado na Figura 5.5 que representa o mesmo script de teste utilizado na Seção 3.4.1 do Capítulo 3 com uma pequena modificação: um dos passos foi considerado prototipado. A modificação introduzida diz respeito ao passo *Step 2.P: Login*. Nós substituímos a chamada da UF `imTk.loginIM()` pela chamada do método `prototype("imTk.loginIM()")` (como pode ser visto no script apresentado na Figura 5.5). Ao fazer tal modificação, estamos considerando que a implementação da UF `LoginIM` não existe para a família do telefone celular em questão (sob teste). Note que o método `prototype` recebe uma String como parâmetro. Mostramos no Capítulo 3 que esse script executa normalmente no TAF até encontrar o método `prototype`. Nesse momento, a execução do script é pausada até que o testador a libere. Esse método é bastante útil durante o desenvolvimento (manual) dos scripts de teste. Nesse aspecto, um script prototipado pode ser usado para inferir uma métrica, pois o desenvolvedor tem uma visibilidade total da quantidade de novas funcionalidades que precisam ser desenvolvidas.

A representação CSP correspondente ao caso de teste prototipado apresentado na Figura 5.5 pode ser visto na Figura 5.7. O `TC_01` é apresentado como uma composição seqüencial do processo `TC_01_Hiding` e o próprio `TC_01`. O processo `TC_01_Hiding` foi criado a fim de agrupar todos os passos de verificação presentes nos scripts de teste, pois ao extrair automaticamente o modelo que representa o comportamento do telefone (`M`), tais verificações não se encontram presentes no modelo. É importante lembrar que `M` é uma visão caixa preta do sistema e que todas as UFs de verificação do TAF não mudam o estado atual do telefone, ou seja, elas existem simplesmente para procurar por um item da tela retornado pelo PTF, mas não executam nenhum tipo de navegação que possa mudar o estado atual do telefone. Por não alterarem o estado atual do telefone, nenhum desses eventos estarão presentes em `M` e por esse motivo foram adicionados ao processo que os esconde (através do uso do operador de CSP *hiding*) ao serem colocados em paralelo com outro processo. Outro ponto que precisa ser discutido é a presença do evento `prototype.1`. Esse evento representa o comando `prototype` do script de teste da Figura 5.5. Especificamente para esse caso de teste, o evento `prototype.1` é o ponto onde devemos nos concentrar. Considerando que temos o modelo extraído do telefone `M`, que pode ser visto na Figura 5.6, o nosso objetivo é encontrar em `M` o evento (ou os eventos) que representam o evento `prototype.1`.

Porém um simples refinamento não resolve o problema em questão, pois devemos considerar que um `prototype` pode ser representado por uma seqüência de eventos. Quando uma relação de refinamento falha, somente um contra-exemplo (onde apenas um evento do modelo onde a falha pode ser observada) é retornado e com isso não conseguiríamos resolver o problema, caso o `prototype` em questão seja uma abstração de mais de um evento do modelo. Além disso, um único `TC` pode ter vários eventos `prototype` a serem resolvidos. Para que possamos resolver todos eles, precisamos atualizar dinamicamente o próprio `TC` com eventos vindos de `M` (enquanto resolvemos cada `prototype`) para continuar com o refinamento até que o evento de terminação (`SKIP`) de `TC` seja encontrado. Além disso, também existe a necessidade de manter um mapeamento de todos os eventos `prototype` com os respectivos eventos de `M` que os resolvem. Todos esses problemas serviram de motivação para desenvol-

```
TC_01 = TC_01_Hiding ; TC_01

TC_01_Hiding = T1 \
{|
   verify.DTCHECK_ITEM.(SCREEN, {IM_APPLICATION}).(DISPLAYED, {}),
   verify.DTCHECK_ITEM.(SCREEN, {CONTACT_LIST}).(DISPLAYED, {}),
   verify.DTCHECK_ITEM.(SCREEN, {CONVERSATION_ITEM}).(DISPLAYED, {}),
   verify.DTCHECK_ITEM.(SCREEN, {IDLE}).(DISPLAYED, {})
|}

T1 = steps ->

start.DTSTA_APPLICATION.(IM_APPLICATION, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {IM_APPLICATION}).(DISPLAYED, {}) ->

prototype.1 ->

verify.DTCHECK_ITEM.(SCREEN, {CONTACT_LIST}).(DISPLAYED, {}) ->

select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN}) ->

start.DTSTA_LISTITEM.(CONVERSATION_ITEM, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {CONVERSATION_ITEM}).(DISPLAYED, {}) ->

exit.DTEXI_APPLICATION.(IM_APPLICATION, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {IDLE}).(DISPLAYED, {}) ->

SKIP
```

Figura 5.7 Especificação CSP de um caso de teste prototipado.

vermos um algoritmo (Algoritmo 1 apresentado no Capítulo 4) com a finalidade de tratar essas particularidades. A seguir esse algoritmo será executado utilizando os modelos apresentados nesse capítulo.

Iniciamos a execução do algoritmo definindo `mapping` como uma sequência vazia. O próximo passo executa o refinamento `TC_M` $\sqsubseteq_F$ `TC`, em que `TC_M` é definido como o paralelismo generalizado entre `TC` e `M` (`TC_M = TC [| Events |] M`). A sequência resultante desse refinamento é armazenada em `tr`. Considerando que estamos executando o algoritmo tomando como base o modelo comportamental apresentado na Figura 5.6 e o caso de teste abstrato apresentado na Figura 5.7, o resultado armazenado em `tr` é a sequência

```
<steps,
 start.DTSTA_APPLICATION.(IM_APPLICATION, {}),
 prototype.1>
```

É importante lembrar que

```
verify.DTCHECK_ITEM.(SCREEN, {IM_APPLICATION}).(DISPLAYED, {})
```

não aparece nessa sequência pelo fato desse evento pertencer ao conjunto de eventos escondidos (`TC_01_Hiding`) do processo `TC_01`.

Passamos então para o próximo passo do algoritmo que é a execução da função `last(tr)`. Para a situação em questão, ela atribui a `pt` o evento `prototype.1`, pois esse é o último evento contido na sequência `tr`. Nesse momento identificamos que `prototype.1` é o problema que precisamos resolver. Observe que a condição do `while(tr` $\neq \langle\rangle$ `and pt` $\neq \checkmark$`)` é satisfeita e então o algoritmo atribui o valor `3`, que representa o tamanho da sequência contida em `tr`, à variável `deep`.

Agora o algoritmo busca na especificação CSP de `System` (apresentado na Figura 5.6) o primeiro evento que deve substituir o evento `prototype.1` em `TC_01`. Para tal, a variável `event` conterá o último evento (`last()`) resultante do contra-exemplo gerado pelo seguinte refinamento

`FindTr(deep)` $\sqsubseteq_F$ `M` $[|\mathit{Events}|]$ `Build(front(tr))`.

Recorde da Seção 4.3, que a função `FindTr()` retorna todos os eventos do alfabeto até o tamanho desejado (passado por parâmetro) e a função `Build()` recebe como parâmetro uma sequência e cria um processo composto pelos elementos dessa sequência.
O contra-exemplo resultante desse refinamento é a sequência

```
<steps,
 start.DTSTA_APPLICATION.(IM_APPLICATION, {}),
 login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {})>
```

Como dito anteriormente, essa sequência é processada pela função `last()` que retorna como resultado o evento

```
login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {})
```

Seguindo o fluxo de execução do algoritmo, a função `getNextEvent()` é executada e retorna como resultado o evento

```
select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN})
```

que por sua vez corresponde ao evento que ocorre imediatamente após `prototype.1` em `TC_01`. O valor desse evento é armazenado na variável `nextTCEvent`.

A condição do `while(event` $\neq$ `nextTCEvent)` (mais interno) também é satisfeita, pois

```
event = login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {})
```

e

```
nextTCEvent = select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN})
```

ou seja, as variáveis envolvidas referenciam valores diferentes. A partir disso, a variável `mapping` passa a conter

```
< (prototype.1,
  <login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {})>) >
```

que é resultado da associação do evento problemático (`prototype.1`) com a seqüência procurada (`login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {})`), criada pela função `updateMapping()`. Note que essa representação indica que o evento `prototype.1` em `TC_01` deve ser substituído pela sua sequência correspondente.

Continuando a execução do algoritmo, é atribuído à variável `i` o valor `3`, que corresponde a soma do valor atual de `deep` com o tamanho atual da sequência correspondente ao evento `prototype.1` armazenado em `mapping`. O próximo passo do algoritmo tem por objetivo encontrar o próximo evento em `M` que possa compor a sequência correspondente ao `prototype.1`. No nosso estudo de caso, o novo valor de `event` é

```
select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN})
```

Nesse momento a condição do `while` mais interno já não é satisfeita, pois os valores referenciados pelas variáveis `event` e `nextTCEvent` são iguais. Ao sair desse laço, a função `updateTC()` é executada. Essa função atualiza o processo `TC_01` (apresentado na Figura 5.7), substituindo o evento `prototype.1` pela sequência de eventos correspondentes ao `prototype.1` em `mapping`. Assim, o novo processo `TC_01` é definido como apresentado na Figura 5.8. Note que o evento `prototype.1` foi substituído por

```
login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {})
```

```
TC_01 = TC_01_Hiding ; TC_01

TC_01_Hiding = T1 \
{|
   verify.DTCHECK_ITEM.(SCREEN, {IM_APPLICATION}).(DISPLAYED, {}),
   verify.DTCHECK_ITEM.(SCREEN, {CONTACT_LIST}).(DISPLAYED, {}),
   verify.DTCHECK_ITEM.(SCREEN, {CONVERSATION_ITEM}).(DISPLAYED, {}),
   verify.DTCHECK_ITEM.(SCREEN, {IDLE}).(DISPLAYED, {})
|}

T1 = steps ->

start.DTSTA_APPLICATION.(IM_APPLICATION, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {IM_APPLICATION}).(DISPLAYED, {}) ->

login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {CONTACT_LIST}).(DISPLAYED, {}) ->

select.DTSEL_LISTITEM.(CONTACT_ITEM, {JOHN}) ->

start.DTSTA_LISTITEM.(CONVERSATION_ITEM, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {CONVERSATION_ITEM}).(DISPLAYED, {}) ->

exit.DTEXI_APPLICATION.(IM_APPLICATION, {}) ->

verify.DTCHECK_ITEM.(SCREEN, {IDLE}).(DISPLAYED, {}) ->

SKIP
```

Figura 5.8 Especificação CSP obtida após substituição do evento *prototype*.

O algoritmo continua a execução dos últimos dois passos. Observe que após a atualização de `TC_01`, o mesmo passou a se comportar exatamente igual ao modelo `System`. Dessa forma, o resultado do refinamento $\texttt{TC_M} \sqsubseteq_F \texttt{TC}$ atribuído à `tr` é uma sequência vazia, pois os processos refinam e, por sua vez, `pt` recebe ✓. Nesse ponto, o algoritmo encerra, pois a condição do laço mais externo não é mais satisfeita. Ao final do algoritmo, o que nos interessa é o conteúdo de `mapping`. Nessa variável estão armazenados todos os *prototypes* do `TC_01` e seus eventos correspondentes em `System` que representam, em CSP, os eventos que identificam aos passos que irão compor o código das novas UFs.

Uma vez que temos em mãos o mapeamento correspondente aos eventos `prototype`, fazemos uma busca nos elementos da estrutura de dados auxiliar mantida pela BxT, que armazena os passos referentes à execução do software embarcado com a finalidade de representar cada um dos eventos do modelo comportamental. Para facilitar o entendimento, criamos a Figura 5.9 que apresenta um conjunto de telas que correspondem aos passos realizados para obtenção do modelo comportamental apresentado na Figura 5.6.

As telas dentro do retângulo vermelho correspondem ao trecho de execução que buscamos. Observe que os passos que nos fazem alcançar essas duas telas representam exatamente os passos que devemos seguir (tanto manual quanto automaticamente) para executar o caso de teste apresentado na Figura 5.5 completamente. Em se tratando desse script TAF, o comando `prototype()` presente em `prototype("imTk.loginIM()")` será removido e nós iremos criar uma nova assinatura de método no *Toolkit* de IM, denominada *loginIM()* para que os scripts de testes tenham visibilidade à essa nova funcionalidade. O conteúdo desse novo método criado é a chamada à mais nova UF API do TAF gerada também automaticamente denominada `LoginIM`. Note que o conteúdo da String passada como parâmetro para o método `prototype()` foi utilizado por nós para definir tanto a nova assinatura no *Toolkit* de IM quanto para definir o nome da nova UF API. O conteúdo da nova implementação da UF `LoginIM` (denominada `LoginIMImp`) será dado pela sequência de comandos enviados pela BxT para o telefone a fim de obter o evento do modelo (apresentado na Figura 5.6) que encontramos após a execução do algoritmo. Esse evento é o que substitui o evento `prototype` do caso de teste abstrato (apresentado na Figura 5.7). Essa sequência de comandos representa justamente a transição entre as telas dentro do retângulo vermelho apresentadas na Figura 5.9. O conteúdo do arquivo `LoginIMImp.java`, que corresponde à implementação da UF `LoginIM`, é apresentado na Figura 5.10. Observe que somente a sequência de comandos enviados pela BxT para o telefone em questão representam a implementação dessa UF. Nesse momento o script de teste já está pronto para ser executado no telefone que gerou o modelo utilizado pelo algoritmo.

Caso o conteúdo da String passada para o método `prototype()` não esteja no padrão *validToolkit.UFName*, um novo método `prototypeX()` será adicionado em um *toolkit* especial denominado *Prototype Toolkit*. Nesse caso, o nome `PrototypeX` é atribuído à nova UF API, que por sua vez terá uma nova implementação denominada `PrototypeXImp`, cujo conteúdo é o mesmo apresentado na Figura 5.10.

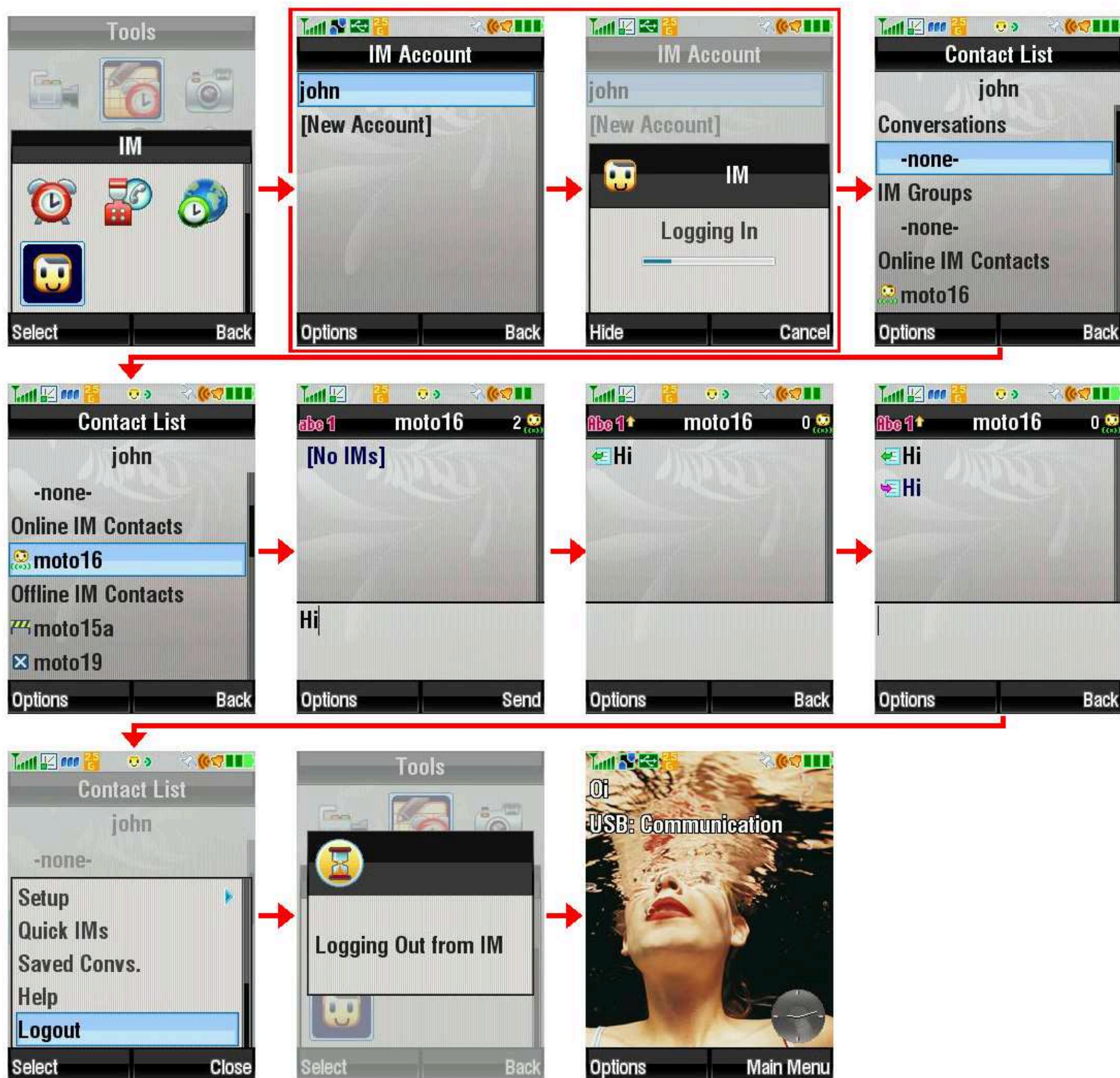


Figura 5.9 Telas referentes à execução do caso de teste apresentado na Figura 5.7.

```
execute()
{
    // Select the IM account to login
    ptf.phone.pressKey(CENTER);

    // Wait until dialog disappear
    ptf.phone.waitDialog(LOGGING_IN);
}
```

Figura 5.10 Conteúdo da implementação da UF LoginIMImp gerada automaticamente.

CAPÍTULO 6

Conclusão

A automação de testes é uma tendência na área de testes de software, tendo como principal objetivo a redução de custos. O *framework* de automação de testes utilizado na Motorola, denominado TAF [KKR+07, Rec06] (*Test Automation Framework*), difere de outros *frameworks* pelo fato dos casos de testes (scripts) serem escritos através da chamada de métodos que representam funcionalidades de alto nível, possibilitando assim customizar essas funcionalidades para diversos produtos. Dessa forma os scripts de teste permanecem imutáveis para todos os produtos de uma mesma família, enquanto as implementações das funcionalidades de alto nível são re-implementadas para se adequarem aos comportamentos específicos dos produtos. O presente trabalho propõe uma forma de desenvolver essas funcionalidades de alto nível do TAF, denominadas UFs (*Utility Functions*), de forma automática, baseado no refinamento entre modelos CSP [RHB97] que representam o comportamento dos telefones, também extraídos de forma automática, e modelos que representam os scripts de teste TAF com chamadas a funcionalidades ainda não implementadas no *framework*.

Uma importante vantagem do uso da abordagem proposta pelo nosso grupo de pesquisa em substituição à utilização do processo tradicional de testes de software diz respeito à melhoria na qualidade (evita a introdução de erros humanos) e produtividade (redução de esforço) como um todo no processo de desenvolvimento de software. A idéia é eliminar alguns passos comuns ao processo de teste de software, como por exemplo escrita e execução de testes, através da substituição por técnicas de teste baseados em modelos (MBT - *Model Based Testing* [EFW01]). Porém, algumas aplicações necessitam observar as características da interação do software sendo executado em um determinado hardware (como é o caso da telefonia celular), para que seja possível obter uma validação completa do sistema sob teste (SUT - *System Under Test*). No trabalho apresentado nessa dissertação, foi considerado esse importante requisito (integração hardware e software) para testar aplicações embarcadas em telefones celulares Motorola.

A estratégia apresentada nessa dissertação é baseada na aplicação de métodos formais para geração de código de UFs para o TAF e não de aplicar métodos formais (validação de modelos) em substituição aos testes. A estratégia proposta requer duas entradas: a primeira é o documento de requisitos escrito em linguagem natural controlada (LNC) e a segunda é o modelo comportamental do telefone celular a ser testado. A primeira entrada é processada pela ferramenta TaRGeT [NCT+07] que gera scripts de teste TAF tanto em Java (concretos) quanto em CSP (abstratos). Mostramos que se a ferramenta não encontrar uma determinada funcionalidade em sua base de dados, durante o seu processamento, um comando especial, denominado *prototype*, é introduzido no script de teste. Para obter a segunda entrada que corresponde ao modelo comportamental do telefone celular a ser testado (em CSP), utilizamos a ferramenta

que desenvolvemos denominada BxT (*Behavior Extractor Tool*) [BM07]. De posse desses dois modelos, nós realizamos uma verificação de refinamento entre eles e assim obtemos uma representação abstrata do conjunto de UFs que precisam ser desenvolvidas para compor o TAF a fim de executar o caso de teste concreto em questão. A ferramenta BxT, além de prover o modelo comportamental do sistema em CSP, também armazena todos os itens relacionados à GUI (*Graphical User Interface*) em uma estrutura de dados, bem como as teclas que foram pressionadas e todos os eventos necessários para que aquela tela ou item pudesse ser alcançado. Os itens da GUI junto com a informação de como chegar a cada um desses itens são suficientes para produzir o código-fonte das UFs, pois uma UF nada mais é do que a execução do sistema através de uma visão caixa-preta do mesmo. Dessa forma, utilizamos os eventos resultantes do refinamento a fim de encontrarmos a implementação das funcionalidades que precisamos resolver e assim atualizamos a base de dados de UFs do TAF automaticamente.

A principal contribuição do presente trabalho foi a proposição de uma forma para desenvolver código para um *framework* de automação de testes a fim de executar scripts de testes também gerados de forma automática diretamente através do processamento de documentos de requisitos. O nosso objetivo inicial foi propor uma abordagem que fosse capaz de substituir o esforço humano no desenvolvimento de UFs relativamente simples. Mostramos, através de um estudo de caso, que abordagem proposta alcançou nossos objetivos, ou seja, mostramos que o objetivo de propor que coisas relativamente simples feitas pelo homem podem ser realizadas pela máquina. Também é importante salientar que as UFs geradas automaticamente através da utilização de nossa abordagem devem passar por processo de inspeção para validar o código gerado antes de ser integrado ao *framework*.

6.1 Trabalhos Relacionados

Testes baseados em especificações formais (modelos) vêm se tornando uma área de pesquisa atrativa, especialmente depois que Gaudel publicou o trabalho intitulado *Testing can be formal too* [Gau95]. O principal motivo do surgimento desse interesse deve-se ao fato de que especificações formais permitem a automação de testes de caixa preta. Nesse sentido, as especificações podem ser analisadas a fim de gerar casos de testes. Além disso, elas fornecem alguns elementos que nos permitem predizer a saída esperada de um caso de teste.

Modelos formais também podem ser utilizados para realizar testes de GUI (*Graphical User Interfaces*). Existem abordagens para geração de casos de teste automaticamente, seleção de casos de teste e utilização de ferramentas para extração de informações a fim de realizar testes de GUI [MBN03, Mem07, XM07]. Memon [MBN03] apresentou uma abordagem para geração de modelos a partir dos *widgets* da GUI, capturados através da execução da aplicação. Nesse contexto, nossa abordagem se assemelha a esse trabalho, porém difere no fato de que usamos o código do *framework* de automação de testes utilizado na Motorola denominado PTF para extrair nossos modelos. Usando o PTF, nós podemos extrair tanta informação quanto precisarmos, enquanto a extração de informações através da GUI é limitada pelo que é oferecido para interação com o usuário. Outra diferença importante está relacionada ao método de teste. Nós utilizamos os modelos obtidos através da execução da BxT [BM07] com o objetivo de gerar um modelo e futuramente descobrir as funcionalidades ainda não implementadas no

TAF, através de verificações de refinamento, a fim de gerar o código para essas funcionalidades automaticamente. Já Memon propõe que o modelo gerado através da extração de informações da GUI seja verificado pelos projetistas de teste e então utilizado para a geração de casos de teste (não automatizados).

Em [AG06] os autores mostraram os problemas do uso de métodos formais em teste e que, como resultado da aplicação da abordagem hipotético-dedutiva, os casos de testes formais podem ser muito abstratos (de baixa qualidade) para garantir a confirmação de uma especificação. Uma outra limitação fundamental da abordagem hipotético-dedutiva está relacionada à não completude de alguns métodos de prova para a relação de refinamento. Considerando o fato de que o uso de métodos formais em substituição à execução de testes de software, procuramos complementar o trabalho de Bertolini [BM07] que apresenta uma abordagem do uso de *model checking* com a finalidade de substituir a aplicação tradicional de testes de software. Nós propomos um método que usa métodos formais com a finalidade de gerar código para um *framework* de automação de testes, sem que haja a necessidade de intervenção humana. A idéia é aumentar a quantidade de funcionalidades fornecidas pelo *framework* a fim de que a interação com os telefones celulares sob teste através da navegação nos menus, seleção de itens em listas, etc.

Em Souza [dS07], é descrito um método para geração de scripts de teste para o TAF através do processamento de LNC dos casos de teste derivados também automaticamente a partir dos documentos de requisitos [dCN06]. O trabalho proposto nessa dissertação é uma extensão do trabalho apresentado em [dS07], visto que, ao se utilizar o método proposto por Souza, onde em um banco de dados relacional associam-se eventos CSP a scripts TAF correspondente, se um evento CSP não for encontrado no banco de dados então a funcionalidade correspondente é simplesmente prototipada no script de teste gerado. Em [dS07], a idéia proposta é que esse comando especial denominado `prototype` seja resolvido futuramente através do esforço dos desenvolvedores do TAF. Nossa abordagem utiliza esses casos de teste prototipados como entrada a fim de criar automaticamente novas implementações de UFs para o TAF que representem as funcionalidades que deveriam ser implementada por desenvolvedores (manualmente), minimizando assim o esforço necessário para realizar essa atividade.

Trabalhos como [DBG01, SA00] usam descrições do sistema para extrair manualmente ou semi-automaticamente um modelo de teste que descreve a arquitetura de um processador. Em [DBG01], o objetivo do projeto MμALT é traduzir manualmente para uma FSM usando GDL (*GOTCHA*[1] *Definition Language*) que forma o modelo de teste [HN99]. GOTCHA é uma ferramenta para geração de um conjunto de testes abstratos a partir de um modelo de teste do SUT e de um grupo de diretivas de teste.

Em [SA00], um modelo de testes de um microprocessador foi construído de forma semi-automática. A parte de controle na descrição inicial do projeto é implementada de forma genérica usando uma FSM que encapsula a descrição do comportamento de controle. Os estados dessa FSM são selecionados naturalmente como candidatos dos estados do modelo de teste. Os autores desenvolveram um algoritmo para extrair esses estados automaticamente a partir de um microprocessador RTL Verilog ou de um projeto VHDL [Ash98].

Abordagens de modelagem que usam especificações em linguagem natural para geração

[1]GOTCHA é um acrônimo para *Generation of Test Cases for Hardware Architectures.*

de um modelo de testes são similares à nossa abordagem. Nesses casos, onde o SUT foi especificado em linguagem natural sem especificações formais ou descrições de implementação, modelos de teste precisam ser construídos manualmente. A principal diferença entre os trabalhos apresentados a seguir e o nosso trabalho diz respeito à abordagem de geração do modelo de testes. No nosso caso, esse modelo é gerado de forma automática enquanto várias outras abordagens necessitam de esforço manual para sua construção. Nessas abordagens, a vantagem de utilizar descrições em linguagem natural é a utilização do alto nível de abstração observado em frases escritas naturalmente (que nos dá uma visão caixa preta do sistema). Detalhes de implementação estão em um nível de abstração mais baixo e, consequentemente, são mais difíceis de entender e de abstrair o comportamento do sistema.

O trabalho proposto em [BFdV+99] tem por objetivo estudar a viabilidade da derivação e execução automática de testes através de um determinado número de especificações formais e de diferentes abordagens para a execução de testes. A geração de testes foi feita com uma ferramenta denominada TorX que é um ambiente de testes baseado em modelos (MBT) genérico. TorX permite a adição de diferentes ferramentas de geração de testes, aceitando inclusive modelos de SUT especificados em diferentes linguagens formais. Nesse trabalho, três modelos formais do mesmo SUT foram gerados. Um em LOTOS [ED89], outro em SDL [M. 91] e o último em PROMELA [Hol03]. Cada um desses foi utilizado com uma ferramenta de teste diferente rodando em um mesmo ambiente (TorX).

O trabalho apresentado em [J. 03] usa uma linguagem de modelagem denominada AutoFocus com a finalidade de gerar um modelo de testes para o WIM (*WAP Identity Module*). AutoFocus é uma ferramenta para desenvolver especificações gráficas para sistemas embarcados baseados em técnicas de descrição concisa e simples. Um modelo em AutoFocus é hierarquicamente organizado em um conjunto de FSMs comunicantes, sincronizadas e que utilizam programas escritos em um paradigma funcional para definição de suas guardas e atribuições.

Alguns modelos de teste nos trabalhos apresentados acima definem estados explicitamente. Na abordagem utilizada pela ferramenta TaRGeT os estados são definidos como parâmetros dos processos nos modelos de teste (em um modelo que segue uma álgebra de processos, como CSP, os parâmetros definem os estados dos processos).

Embora não definamos estados de forma explícita em nossos modelos de teste, nós podemos caracterizar um estado como um trace da execução do modelo. Assim, considerando o ponto inicial da execução é possível definir o estado do modelo seguindo a execução do trace no modelo. É importante salientar que nós consideramos essa notação de estado como implícita porque nosso modelo de teste não define estados explicitamente.

O processo de geração de casos de teste a partir dos modelos de teste tem como resultado a geração de casos de teste abstratos (especificações formais), sendo assim muito difícil para um testador sem conhecimento na linguagem de especificação formal utilizada executar esses casos de teste diretamente no SUT. Para resolver esse problema é necessário traduzir esses casos de teste abstratos em casos de teste concretos (escritos usando LNC). Por esse motivo, a estratégia MBT utiliza um componente (denominado *tc-translator* [PERH05]) que traduz os casos de teste abstratos em casos de teste concretos, podendo esses últimos serem aplicados diretamente para realização de testes no SUT.

O *tc-translator* tem a função de fazer a ligação entre o modelo de teste e o SUT através

da adição de informações e da tradução das entidades do caso de teste abstrato em construções concretas para a linguagem da plataforma de testes a ser utilizada. Por exemplo, se o valores do tipo de dado de um operando são abstraídos em classes equivalentes no modelo de teste, o *tc-translator* seleciona um operando concreto para aquela operação [SA00]. No nosso caso, os operandos são eventos do sistema. Assim nos casos de teste abstratos apresentados nessa dissertação os eventos são definidos em CSP, enquanto nos casos de teste concretos os eventos são traduzidos em comandos que juntos compõem um script TAF.

Algumas abordagens MBT usam um arquivo de configuração ou tabela para configurar o *tc-translator* [J. 03, FHP02, SA00]. Essa tabela contém as relações de transição entre entidades abstratas (estados, operações e outros) do modelo e instruções concretas para a plataforma de testes do SUT. Dessa forma o *tc-translator* pode ser ajustado facilmente para diferentes plataforma de testes. Frequentemente uma operação no modelo é uma macro para o *tc-translator* e é substituído por várias instruções no nível concreto. A modelagem apresentada em [dS07] e também utilizada nessa dissertação segue a mesma idéia. Inicialmente é obtido um casos de teste concreto e depois o mesmo é traduzido para casos de testes abstratos. Assim, através da tradução de casos de testes concretos em abstratos, nós armazenamos o mapeamento de entidades entre eventos abstratos e concretos em um banco de dados relacional.

6.2 Trabalhos Futuros

Embora a versão atual da ferramenta BxT seja funcional, ela tem algumas limitações: não é possível obter informações sobre comportamentos que precisam de dados específicos de entrada (por exemplo, telas que requerem senhas), sobre informações relacionadas a estados internos do telefone, por exemplo que representam que a memória de mensagens está cheia ou que a caixa de e-mails está vazia, e assim sucessivamente. Além disso, requisitos não funcionais como desempenho e componentes visuais (como qualidade de imagens) não são capturados. Consequentemente, como trabalho futuro, nós pretendemos melhorar o mecanismo da BxT a fim de cobrir essas situações, pois a qualidade das UFs geradas dependem diretamente da qualidade do modelo gerado pela BxT. Porém requisitos relacionados a interações externas (como remoção de bateria ou cartão de memória) somente são possíveis através de intervenção humana (manual).

A abordagem utilizada para obtenção de resultados nesse trabalho não considera a parametrização das funcionalidades geradas. Assim, um trabalho que agregaria um valor interessante ao método proposto seria a geração de UFs parametrizadas. Dessa forma seria possível fazer o reuso de funcionalidades já existentes, como por exemplo, parametrizar uma UF que tem por objetivo navegar entre os contatos do *phonebook* a fim de encontrar um contato específico seria interessante, pois se fosse necessário buscar outro contato, a mesma implementação poderia ser utilizada, sendo modificado apenas o parâmetro passado para a UF. Para tanto será necessário modificar a abordagem de modelagem utilizada atualmente e considerar a noção de estados na modelagem CSP.

Além disso, manter uma base de dados das UFs existentes no TAF seria bastante interessante, pois conseguiríamos reusar essas UFs durante a geração de implementações para as UFs. Ao considerar o reuso do próprio TAF no desenvolvimento, é possível obter um código ainda

mais próximo do código gerado por desenvolvedores experientes. Identificamos também que algumas UFs geradas não funcionaram corretamente devido à incompatibilidade das pré e/ou pós condições das UFs já existentes no TAF com as UFs geradas automaticamente. As pré e pós condições das UFs representam respectivamente o estado do telefone antes e depois de executar cada uma delas. Esse é mais um motivo de utilizar a abordagem de especificar formalmente a cada UF do TAF, considerando os seus estados (parâmetros).

Mostramos também que o algoritmo proposto tem algumas limitações que precisam ser resolvidas. Já visualizamos algumas soluções que precisam ser implementadas, como por exemplo, considerar a geração semi-automática do código nos casos em que o modelo disponibiliza mais de uma opção de implementação para uma determinada funcionalidade. Nesses casos, a ferramenta deve mostrar as opções para que o usuário escolha qual dessas opções se apresenta como melhor solução para o problema. No que diz respeito a dois ou mais eventos *prototype* consecutivos, a solução seria considerar eles como somente um *prototype* e executar normalmente o algoritmo proposto.

Por fim, identificamos mais três trabalhos futuros: o primeiro seria o desenvolvimento de uma interface gráfica para auxiliar tanto na execução do método proposto quanto para facilitar a integração do método na ferramenta TaRGeT; o segundo seria investigar a geração de código para outros *frameworks* de automação de testes utilizados na Motorola a fim de fazer uma análise comparativa do método proposto; e o terceiro trabalho seria realizar uma análise no que diz respeito ao desempenho da execução do código gerado (gerar mesmas UFs automática e manualmente a fim de analisar o tempo e a qualidade da execução), bem como a cobertura do método proposto.

Referências Bibliográficas

[AB07a] Eduardo Aranha and Paulo Borba. An Estimation Model for Test Execution Effort. In *ESEM'07: International Symposium on Empirical Software Engineering and Measurement*, page 9, Madrid, Spain, 2007.

[AB07b] Eduardo Aranha and Paulo Borba. Estimation Model for Test Execution Effort. In *ICSE'2007: 29th International Conference on Software Engineering*, page 7, Minneapolis, MN, USA, 2007.

[ABL06] Eduardo Aranha, Paulo Borba, and Jose Lima. Model Simulation for Test Execution Capacity Estimation. In *17th International Symposium on Software Reliability Engineering (ISSRE 06)*, pages 6–11, Raleigh, NC, USA, 2006.

[AG06] Bernhard K. Aichernig and Chris George. When Model-based Testing Fails. *Electronic Notes Theoretical Computer Science*, 164(4):115–128, 2006.

[Apa06] Apache. JMeter - Apache JMeter. http://jakarta.apache.org/jmeter/, Aug 2006.

[Ash98] P. Ashenden. *The Student's Guide to VHDL.* Morgan Kaufmann Publishers Inc, 1st edition, 1998.

[Bac99] J. Bach. Test Automation Snake Oil. In *14th International Conference and Exposition on Testing Computer Software*, Washington, DC, 1999.

[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. *ACM Computing Surveys*, Dec 2003.

[Bei95] Boris Beizer. *Black-Box Testing*. John Wiley & Sons, 1995.

[BFdV+99] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal Test Automation: A Simple Experiment. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 179–196, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[BIMP04] A. Bertolino, P. Inverardi, H. Muccini, and A. Polini. Towards Anti-Model Based Testing. In *Proceedings of Fast Abstract Session at IEEE International Conference on Dependable Systems and Networks DSN 2004*, pages 124–125, 2004.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BM07] Cristiano Bertolini and Alexandre Mota. Using Refinement Checking to Aid Software Testing: An Industrial Experience. In *Submitted to 22nd International Conference on Automated Software Engineering (ASE)*, Atlanta, Georgia, USA, 2007. IEEE Computer Society Press and ACM Press.

[Car06] Emanuela Cartaxo. Test Case Generation by means of UML Sequence Diagrams and Label Transition System for Mobile Phone Applications. Master's thesis, Federal University of Campina Grande (UFCG), 2006.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. Addison-Wesley Longman Publishing Co., Inc., MIT Press, Cambridge, MA, USA, 1999.

[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, Sep 1998.

[Com06a] Compuware. Compuware QADirector – Advanced Risk-based Test Management for Distributed Applications. http://www.compuware.com/products/qacenter/qadirector.htm, Aug 2006.

[Com06b] Compuware. File-AID/CS - Complete Test Data Management for Distributed Environments. http://www.compuware.com/products/fileaid/cs.htm, Aug 2006.

[CS06] Gustavo Cabral and Augusto Sampaio. Formal Specification Generation from Requirement Documents. In *Proceedings of III Brazilian Symposium on Formal Methods*, pages 217–232, Natal, RN, Brazil, 2006.

[DBG01] J. Dushina, M. Benjamin, and D. Geist. Semi-formal test generation with Genevieve. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 617–622, New York, NY, USA, 2001. ACM Press.

[dCN06] Sidney de Carvalho Nogueira. Geração Automática de Casos de Teste CSP Orientada por Propósitos (in Portuguese). Master's thesis, Informatics Center, Federal University of Pernambuco (UFPE), 2006.

[dFLC06] Gustavo da Fonseca Limaverde Cabral. Formal Specification Generation from Requirement Documents. Master's thesis, Informatics Center, Federal University of Pernambuco (UFPE), 2006.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based Testing in Practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[dOGF06] Jorge Antonio Correa de Oliveira, Cidinha Costa Gouveia, and Romulo Souto Quidute Filho. A Test Automation Viability Analysis Method. In *LATW2006: Proceedings of the 7th IEEE Latin American Test Workshop*, page 6, Buenos Aires, Argentina, 2006. IEEE Computer Society Press.

[dS07] Clélio Feitosa de Souza. Modelling and Integrating Formal Models: from Test Cases and Requirements Models. Master's thesis, Informatics Center, Federal University of Pernambuco (UFPE), 2007.

[ED89] P. Van Eijk and Michel Diaz. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989.

[EFW01] I. K. El-Far and J. A. Whittaker. *Model-Based Software Testing*. Wiley-Interscience, Encyclopedia of Software Engineering, second edition edition, 2001.

[Ele90] Electrical, Institute O. and (IEEE), Electronics E. *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[EV06] I. Esipchuk and D. Validov. PTF-based Test Automation for JAVA Applications on Mobile Phones. In *Proceedings of the IEEE 10th International Symposium on Consumer Electronics (ISCE)*, pages 1–3, 2006.

[FdCD+04] Marcelo Fantinato, Adriano C. R. da Cunha, Sindo V. Dias, Sueli A. Mizuno, and Cleida A. Q. Cunha. AutoTest – Um Framework Reutilizável para a Automação de Teste Funcional de Software. In *Anais do III Simpósio Brasileiro de Qualidade de Software*, page 15, Brasília, DF, Brazil, 2004.

[FHP02] E. Farchi, A. Hartman, and S. S. Pinter. Using a Model-Based Test Generator to Test for Standard Conformance. *IBM Systems Journal*, pages 41(1):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.

[GM04] Glenford and J. Myers. *The Art of Software Testing*. John Wiley and Sons, New Jersey, 2004.

[Gra97] M. Grand. *Java Language Reference*. O'Reilly, 2nd edition, 1997.

[Hen98] E. Hendrickson. The Differences Between Test Automation Success And Failure. In *Proceedings of STAR West*, 1998.

[HN99] A. Hartman and K. Nagin. TCBeans, software test toolkit. In *In Proceedings of the 21th International Software Quality Week (Wq 1999)*, pages 445–450, 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Hol03] G. J. Holzmann. *The Spin Model Checker*. Pearson Education, 2003.

[How06] Rick Hower. Software QA and Testing Resources Center - Software QA and Testing Frequently Asked Questions, Part 1. http://www.softwareqatest.com/qatfaq1.html, Aug 2006.

[IBM06] IBM. IBM Software – Rational Robot – Product Overview. http://www-306.ibm.com/software/awdtools/tester/robot/, Aug 2006.

[Int06] IntelliJ. IntelliJ IDEA :: The Most Intelligent Java IDE. http://www.jetbrains.com/idea/, Aug 2006.

[J. 03] J. Philipps and A. Pretschner and O. Slotosch and E. Aiglstorfer and S. Kriebel and K. Scholl. Model-based Test Case Generation for Smart Cards. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03). (Electronic Notes in Theoretical Computer Science, vol. 80)*, pages 168–182, Trondheim, Norway, 2003. Elsevier.

[Jak06] Jakarta. Jakarta Cactus. http://jakarta.apache.org/cactus/, Aug 2006.

[Jor95] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, second edition edition, 1995.

[JUn06] JUnit.org. JUnit, Testing Resources for Extreme Programming. http://www.junit.org/, Aug 2006.

[Kan97] C. Kaner. Improving the Maintainability of Automated Test Suites. *Proceedings of the Thenth International Quality Week*, 4(4), 1997.

[Kat05] Joost-Pieter Katoen. *Labelled Transition Systems*, volume 3472/2005 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.

[KKR+07] Luiz Kawakami, Andre Knabben, Douglas Rechia, Denise Bastos, Otavio Pereira, Ricardo P. Silva, and Luiz Santos. A Test Automation Framework for Mobile Phones. In *Proceedings of the 8th IEEE Lating American Test Workshop (LATW)*, pages 1–8, Cuzco, Peru, 2007.

[Kru00] Philippe Kruchten. *The Rational Unified Process An Introduction*. Addison-Wesley, second edition edition, 2000.

[Lei06] Daniel Almeida Leitão. NLForSpec: Uma Ferramenta para Geração de Especificações Formais a partir de Casos de Teste em Linguagem Natural (in Portuguese). Master's thesis, Informatics Center, Federal University of Pernambuco (UFPE), 2006.

[Lew00] William E. Lewis. *Software Testing and Continuous Quality Improvement*. Auerbach, 2000.

[M. 91] M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing*, 3(1):21–57, 1991.

[Mar97] Brian Marick. Classic Testing Mistakes. In *STAR Conference*, May 1997.

[Mar03] Robert Cecil Martin. *UML for Java Programmers*. Prentice Hall, 2003.

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 260, Washington, DC, USA, 2003. IEEE Computer Society.

[Mem07] Atif M. Memon. An Event-Flow Model of GUI-Based Applications for Testing. *Software Testing, Verification and Reliability*, 2007.

[Mer06a] Mercury. Performance Monitor – Mercury LoadRunner Performance Monitor. http://www.mercury.com/us/products/performance-center/loadrunner/monitors/, Aug 2006.

[Mer06b] Mercury. Regression Testing – Mercury WinRunner. http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/, Aug 2006.

[Mye04] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 2nd edition, 2004.

[NB07] Euclides N. Arcoverde Neto and Cristiano Bertolini. Software Testing Through Refinement Checking. Technical report, Federal University of Pernambuco. Motorola's Collaboration Environment. CInBTCRD Project: https://compass.motorola.com/cgi/go/220564249, 2007.

[NCT+07] Sidney Nogueira, Emanuela Cartaxo, Dante Torres, Eduardo Aranha, and Rafael Marques. Model Based Test Generation: An Industrial Experience. In *Accepted to appear in 1st Brazilian Workshop on Systematic and Automated Software Testing*, João Pessoa, PB, Brazil, 2007.

[Par06] Parasoft. JTest: Java Unit Testing. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, Aug 2006.

[PERH05] W. Prenninger, M. El-Ramly, and M. Horstmann. Case Studies. In M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-based Testing of Reactive Systems: Advanced Lectures*, number 3472 in LNCS. Springer-Verlag, 2005.

[Pre04] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, 2004.

[Que99] Geoff Quentin. Automated Software Testing: Introduction, Management and Performance. *Softw. Test., Verif. Reliab.*, 9(4):283–284, 1999. Elfriede Dustin, Jeff Rashka and John Paul, Addison-Wesley, 1999 (Book Review).

[Rat06] Rational. Using Rational Pure Coverage. http://bmrc.berkeley.edu/purify/docs/html/installing_and_gettingstarted/3-pureCov.html, Aug 2006.

[Rec06] Douglas Rechia. Especificação Formal de Restrições de Projeto para Frameworks Orientados a Objetos (in Portuguese). Master's thesis, Computer Science Departament, Federal University of Santa Catarina (UFSC), 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.

[SA00] Jian Shen and Jacob A. Abraham. An RTL Abstraction Technique for Processor Microarchitecture Validation and Test Generation. *J. Electron. Test.*, 16(1-2):67–81, 2000.

[SAV+05] Augusto Sampaio, Carlos Albuquerque, João 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.

[Sca98] J. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, Oxford University Computing Laboratory, 1998.

[Sch92] S. A. Schneider. An Operational Semantics for Timed CSP. In *Proceedings Chalmers Workshop on Concurrency*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.

[Soa06] Elifrancis Rodrigues Soares. Processo de Análise de Cobertura Alinhado ao Processo de Desenvolvimento de Software (in Portuguese). Master's thesis, Informatics Center, Federal University of Pernambuco (UFPE), 2006.

[Sof99] Rational Software. The Rational Approach to Automated Testing. In *A White Paper From Rational Software Corporation*, page 15, 1999.

[Sof06] Quest Software. Java Profiler for J2EE and Java Performance Monitoring by Quest Software. http://www.quest.com/jprobe/, Aug 2006.

[Som01] Ian Sommerville. *Software Engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Sys03] Formal Systems. *ProBE User Manual*. Formal Systems (Europe) Ltd, 2003.

[Sys05] Formal Systems. *Failures-Divergence Refinement - FDR2 User Manual*. Formal Systems (Europe) Ltd, Jun 2005.

[Tor06] Dante Gama Torres. SpecNL: Uma Ferramenta para Gerar Descrições em Linguagem Natural a partir de Especificações de Casos de Teste (in Portuguese). Master's thesis, Informatics Center, Federal University of Pernambuco (UFPE), 2006.

[XM07] Qing Xie and Atif M. Memon. Designing and Comparing Automated Test Oracles for GUI-Based Software Applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.