UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MAYKON LUÍS CAPELLARI

Reutilização de teste em linha de produtos de software baseado em máquina de estados finitos para sistemas embarcados

Maringá
2012

MAYKON LUÍS CAPELLARI

Reutilização de teste em linha de produtos de software baseado em máquina de estados finitos para sistemas embarcados

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientadora: Profa. Dra. Itana Maria de Souza Gimenes

Maringá
2012

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

C238r

Capellari, Maykon Luís
Reutilização de teste em linha de produtos de software baseado em máquina de estados finitos para sistemas embarcados / Maykon Luís Capellari. -- Maringá, 2012.
76 f. : il. (algumas col.), figs., tabs.

Orientadora: Prof.ª Dr.ª Itana Maria de Souza Gimenes.
Dissertação (mestrado) - Universidade Estadual de Maringá, Centro de Tecnologia, Departamento de Informática, Programa de Pós-Graduação em Ciência da Computação, 2012.

1. Linha de produto de software. 2. Software - Teste. 3. Software - Reutilização de teste. 4. Máquina de estados finitos (MEF). 5. Sistemas embacados. I. Gimenes, Itana Maria de Souza, orient. II. Universidade Estadual de Maringá. Centro de Tecnologia. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22.ed. 005.14
AMMA-00347

FOLHA DE APROVAÇÃO

MAYKON LUÍS CAPELLARI

Reutilização de teste em linha de produtos de software baseado em máquina de estados finitos para sistemas embarcados

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação pela Banca Examinadora composta pelos membros:

BANCA EXAMINADORA

Profa. Dra. Itana Maria de Souza Gimenes
Universidade Estadual de Maringá – DIN/UEM

Profa. Dra. Elisa Hatsue Moriya Huzita
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Marcelo Fantinato
Universidade de São Paulo – EACH/USP

Aprovada em: 02 de março de 2012.
Local da defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá

DEDICATÓRIA(S)

Dedico esse trabalho a meus pais, Deoclecio e Jacira, e a minha esposa Érika, obrigado por tudo.

AGRADECIMENTO(S)

Agradeço primeiramente a Deus que me guiou e deu forças durante essa dura jornada.

Aos meus pais, Deoclecio e Jacira por terem me dado a vida, educação e estudo, pois graças a eles cheguei até aqui, obrigado por tudo, amo vocês.

A minha esposa Érika pela paciência, compreensão, apoio e dar-me forças em todos os momentos principalmente nos difíceis, te amo demais amor.

A toda minha família. Meu irmão Júlio, meus Avós, tios, primos agradeço de coração por terem me apoiado e incentivado durante todo este período.

Agradeço em especial minha orientadora Itana, por ter me aceito como seu orientando, pelos conselhos e broncas, que me ajudaram a ser um profissional melhor. Agradeço pela paciência, compreensão, pelo profissionalismo e disposição, pois mesmo passando por dificuldades sempre me auxiliou e me orientou, obrigado de coração.

Agradeço também ao professor Ades, que se dispôs a ser meu co-orientador, me acolheu, auxiliou e aconselhou durante todo o período do sanduíche e mesmo após este, não me abandonou. Agradeço pela paciência e ensinamentos, graças a você consegui dar um rumo a minha dissertação.

A todos os funcionários e professores do DIN-UEM, em especial aos professores(as): Itana, Elisa, João Angelo, Ronaldo, Renato Balancieri, Tânia e Edson. Agradeço também a Maria Inês Davanço, pela amizade, paciência e por ter me ajudado durante todo este período.

Aos amigos que conheci no LabEs por me acolher, ajudar e pelas risadas: Nardi, Marcão, Fabiano, Bruno, Frotinha, VinnyBoy, Lucas, Neiza, Adriano, Ceará, e todos que direta ou indiretamente contribuiram com meu trabalho. Agradecimento especial ao Endo que durante todo este período me ajudou tirando dúvidas, na correção do trabalho e sempre esteve a disposição para me auxiliar, Valeu Endo!

Agradeço o pessoal do Intermedia, David, Watinha e especial o Tim por me acolher em sua casa durante o sanduíche.

Agradeço especialmente à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo apoio financeiro despendido a este trabalho.

Reutilização de teste em linha de produtos de software baseado em máquina de estados finitos para sistemas embarcados

RESUMO

Linha de produto de software (LP) é uma abordagem recente que oferece diversas vantagens para as organizações, tais como: reduções significativas no desenvolvimento e custo de manutenção e diminuição do *time-to-market*. Muitas vezes o processo de teste não consegue acompanhar a velocidade de desenvolvimento de uma LP. O custo relativo do teste torna-se superior comparado com testes em sistemas tradicionais. Devido a isso, testes em LP consomem muito tempo e esforços ultrapassando geralmente mais da metade do custo de desenvolvimento de sistemas únicos. Existem diversas abordagens de teste que apóiam o gerenciamento e a reutilização de testes de LP. Entretanto, as mesmas não promovem a reutilização de testes, e seus resultados, entre os produtos instanciados da LP. Este trabalho propõe o desenvolvimento de um estratégia de teste incremental baseada em Máquina de Estados Finitos (MEFs), que permite a reutilização de testes entre produtos instanciados de uma LP. Esta reutilização inclui processos, artefatos e resultados de testes já realizados. Visando analisar a viabilidade da estratégia proposta foram realizados dois experimentos, em ambos os casos a estratégia mostrou-se bastante efetiva na geração e reutilização de conjuntos de testes entre os produtos da LP.

Palavras-chave: Linha de Produto. Teste. Reutilização de teste. MEF.

Test reuse in software product line based on finite state machines for embedded systems

ABSTRACT

Software Product Line (PL) is a recent approach which offers several benefits for organizations, such as significant reductions in development and maintenance costs and reduced time-to-market. The testing process cannot always follow the development speed of a PL. Its relative cost becomes higher compared to the testing traditional systems. Thus, tests consume a lot of time and effort usually represent more than half the cost of developing single systems. There are various testing approaches that support the management and reuse of tests in PL. However, they do not promote the reuse of tests, and their results, among the products instantiated from a PL. Thus, this project proposes the development of an incremental test strategy based on Finite State Machines (FSM), this allows the reuse of tests between products instantiated from a PL. This reuse includes test process, artifacts and results already carried out. Aiming to analyze the feasibility of the proposed strategy two experiments were conducted, in both cases the strategy proved very effective in generating and reuse of testing among the products of PL.

***Keywords*:** Product Line. Testing. Reuse Testing. FSM.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AGM	*Arcade Game Maker*
DSDs	*Domain-Specific Descriptions*
DSL	*Domain Specific Language*
FAST	*Family-Oriented Abstraction, Specification and Translation*
FODA	*Feature-Oriented Domain Analysis*
FSM	*Finite State Machine*
LP	Linha de Produto de Software
MEF	Máquina de Estado Finita
PLUCs	*Product Line Use Cases*
PLUS	*Product Line UML-Based Software Engineering*
PLUTO	*Product Line Use Case Test Optimisation*
PuLSE	*Product Line Software Engineering*
SAT	*Satisfiability Problem*
SEI	*Software Engineering Institute*
SMS	*Short Message Service*
TBM	Teste baseado em Modelos
UML	*Unified Modeling Language*

SUMÁRIO

Capítulo

1

Introdução

1.1 Considerações Iniciais

A insatisfação de clientes com produtos de software que não atendem as suas necessidades e não possuem flexibilidade, levou a criação de uma nova maneira de aumentar a produção, oferecer novas escolhas, criar produtos diferenciados e, por fim, reduzir custos. Assim, foi proposta uma nova abordagem de desenvolvimento de software, denominada, Linha de Produto de Software (LP).

Uma LP é definida como uma família de sistemas de software que compartilham um conjunto comum e gerenciável de características que satisfazem as necessidades específicas de um segmento particular de mercado ou missão (Clements e Northrop, 2001). Os produtos membros de uma família são derivados de uma configuração específica de um núcleo de artefatos ([1]). Uma LP é baseada em dois conceitos principais: personalização em massa e plataformas comuns (Pohl et al., 2005). A personalização em massa é a produção em grande escala de produtos adaptados às necessidades de clientes individuais, enquanto plataformas comuns formam uma base de recursos a partir da qual outras tecnologias ou processos são construídos.

A utilização da abordagem de LP nas organizações produz diversas vantagens, tais como: melhoria de produtividade, melhoria da qualidade do produto, diminuição do custo, diminuição do *time-to-market* e aumento da reutilização de componentes de software, entre outras (SEI, 2011).

[1] *Core Assets*

Para garantir a qualidade dos produtos instanciados de uma LP, há a necessidade de uma abordagem de teste que forneça confiança na exatidão de um produto por meio da execução de seu código ou parte deste (Tevanlinna et al., 2004). O teste em LP utiliza técnicas de teste aplicadas em sistemas únicos, tais como teste unitário, teste de integração e teste de regressão, porém o processo de teste é aplicado em duas etapas: na engenharia de domínio (criação do núcleo de artefatos) e na engenharia de aplicação (instanciação de um produto da LP).

Devido à complexidade e ao grande número de possíveis configurações dos produtos de uma LP, geralmente, os testes na engenharia de domínio são realizados apenas nas características comuns e aquelas que serão mais utilizadas. Na engenharia de aplicação são realizados os testes de sistemas, integração e regressão para garantir a qualidade do produto final conforme as características selecionadas (Pohl e Metzger, 2006; Tevanlinna et al., 2004).

Com a adoção de métodos baseados em UML para o desenvolvimento de LPs (Gomaa, 2004), a presença de modelos é um importante fator que deve ser notado na escolha de técnicas de testes. Testes Baseados em Modelos (TBM) é uma abordagem que utiliza os modelos de software existentes para apoiar a geração de casos de teste (Dalal et al., 1999). Modelos de Estados, como Máquinas de Estados Finitos (MEFs), são frequentemente usados para gerar casos de teste que exercitam seqüências de eventos no software. Teste baseado em MEFs tem sido aplicado em vários domínios, incluindo protocolos de redes e Sistemas Embarcados (Broy et al., 2005).

Segundo Pressman (2005), existem diversas categorias de produto de software, tais como software básico, sistemas de informação, sistemas científicos, sistemas embutidos, sistemas pessoais, sistemas de inteligência artificial e sistemas reativos. Sistemas reativos são definidos pela constante interação com ambientes externos, podendo ser com um usuário, um dispositivo de entrada, uma outra parte do sistema, ou mesmo um outro sistema (Furbach, 1993). Sistemas reativos são empregados em atividades que exigem alto grau de qualidade e tolerância a falhas, de maneira que a ocorrência destas em determinados sistemas podem ser catastróficas. Portanto, é necessário que seja realizada atividades de garantia de qualidade durante todo o processo de desenvolvimento.

Atualmente, muitos sistemas embarcados são conhecidos como sendo reativos, por exemplo, eles respondem a eventos (entradas) e produzem ações observáveis (saídas). MEFs tem sido utilizadas para modelar estes sistemas, sendo estes desenvolvidos para aviões, telefones celulares, e dispositivos domésticos.

1.2 Contexto e Motivação

Existem diversas técnicas de teste em LP dentre as quais podemos citar: Reutilização de Testes baseados em Modelos (Olimpiew e Gomaa, 2009); Testes baseados em Caso de Uso de LP (Bertolino e Gnesi, 2003); O Modelo W para teste em LP (Jin-hua et al., 2008); Abordagem de Teste baseado em Especificação para LP (Uzuncaova et al., 2007); Teste em LP usando Geração de Teste Incremental (Uzuncaova et al., 2008); Definição de Caso de Teste a partir de Casos de Uso especificados por uma DSL (Im et al., 2008), apesar de algumas destas abordagens permitirem a reutilização de testes, estas não apoiam a reutilização de testes entre os produtos instanciados da LP, sendo este um desafio no processo de teste em LP. Desta forma, há a necessidade de uma estratégia que apoie esta reutilização garantindo a qualidade dos produtos e da LP. No contexto deste trabalho o termo "reutilização de teste" consiste da reutilização de processos, artefatos e resultados dos testes (resultado da execução dos testes entre os produtos da LP).

Segundo Hierons et al. (2009), a presença de modelos formais e especificações pode fazer com que a atividade de teste seja mais efetiva, além de facilitar a automatização do processo. Várias técnicas formais de especificação foram propostas com o objetivo de facilitar a especificação do comportamento dos sistemas. A utilização de MEFs (Gill, 1962) para a especificação desses sistemas auxilia na automação do processo de teste de conformidade e se destaca pela simplicidade, pelo nível de representação e cobertura de falhas.

A partir da especificação do sistema, podem ser construídas implementações e avaliadas sua corretude através da geração de conjunto de casos de testes. Diversos métodos de geração de casos de testes com MEF vêm sendo propostos há décadas, como por exemplo, os métodos DS (Gonenc, 1970), W (Chow, 1978), UIO (Sabnani e Dahbura, 1988), UIOv (Vuong, 1989), Wp (Fujiwara et al., 1991), HIS (Petrenko et al., 1994) (Luo et al., 1994). Mais recentemente, destacam-se também os métodos H (Dorofeeva et al., 2005), *State Counting* (Petrenko e Yevtushenko, 2005), *Checking Sequence with Reset Transitions* (Hierons e Ural, 2010) e P (Simão e Petrenko, 2010). Utilizando algum destes métodos, são gerados sequências de testes que servem como entradas para as implementações. As sequências de testes são utilizadas para confrontar as saídas obtidas do produto final com a especificação. Erros são detectados pela produção de saídas inconsistentes as definidas pelo modelo testado.

Este projeto de mestrado está inserido no contexto do projeto de pesquisa Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos (INCT-SEC[2]), que inclui a elaboração de uma estratégia de testes para LPs no domínio de Sistemas Embarcados. O INCT-SEC realiza pesquisas na área de Sistemas Embarcados Críticos e envolve grupos de pes-

[2]Site do projeto INCT-SEC disponível em http://www.inct-sec.org

quisadores de doze instituições universitárias brasileiras, entre elas EESC/USP, ICMC/USP, UNESP-Rio Preto, POLI/USP, PUCRS, UEM, UFAM, UFG, ITA e UFSCar, e representantes da iniciativa privada.

1.3 Objetivos

Este trabalho tem como objetivo principal apresentar uma estratégia de teste incremental para LP, que permita a realização de testes em produtos instanciados de uma LP. A estratégia apresentada foi denominada *FSM-TSPL - FSM-based Testing of Software Product Line*.

A estratégia desenvolvida aborda a reutilização de casos de testes baseado em MEFs gerados a partir de produtos instanciados de uma LP. As diferenças entre dois modelos MEFs (dois produtos da LP) são identificadas, e utilizando um método de geração de testes incrementais, apenas novas características são testadas.

Para avaliar e garantir a viabilidade de utilização da estratégia foram realizados dois experimentos com as LPs AGM e Mobile Media, ambas para sistemas embarcados.

1.4 Organização do Trabalho

A seguir esta dissertação está organizada da seguinte maneira:

No Capítulo 2 são apresentados os principais conceitos sobre teste de software. São apresentados conceitos sobre Linha de produto de software e principais métodos de teste no Capítulo 3. A estratégia é apresentada no Capítulo 4. A condução dos experimentos e resultados são apresentados no Capítulo 5 e as conclusões e trabalhos futuros são apresentados no Capítulo 6.

Capítulo

2

Teste de Software

2.1 Considerações Iniciais

Neste Capítulo, são abordados conceitos de teste de software, bem como técnicas para garantir a qualidade do software. São considerados fundamentos, técnicas, critérios e formas de avaliação de abordagens de teste de software.

Este capítulo está organizado da seguinte forma. Na Seção 2.2 são definidos termos e conceitos de teste de software. Na Seção 2.3, são apresentados os tipos de critérios de teste e técnicas de teste. Na Seção 2.4, são apresentados alguns modelos formais usados no teste baseado em modelos. Métodos de geração de casos de teste são apresentados na Seção 2.4.1. Por fim, na Seção 2.5 são apresentadas as considerações finais deste capítulo.

2.2 Conceitos e Definições

As técnicas de teste são usadas para identificar defeitos durante a construção de produtos com o objetivo de garantir que, após concluídos, eles possuam as qualidades inicialmente especificadas. Existem diversas definições para a palavra "teste", porém muitas destas são incorretas, pois descrevem quase o oposto de como os testes devem ser vistos. A definição mais apropriada seria "Teste é um processo de executar um programa com a intenção de encontrar erros" (Myers e Sandler, 2004).

No contexto de teste de software, o padrão IEEE 610.12 (IEEE, 1990) diferencia os seguintes termos: defeito (*fault*) - passo, processo ou definição de dados incorreta (instrução ou comando incorreto); engano (*mistake*) - ação humana que produz um resultado incorreto

(uma ação incorreta tomada pelo desenvolvedor); erro (*error*) - a diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução; e falha (*failure*) - produção de uma saída incorreta em relação à especificação, ocorre em conseqüência de um erro, defeito ou engano. O padrão também define o conceito de caso de teste como um conjunto de entradas de teste, condições de execução e resultados esperados criados para um objetivo particular como exercitar um determinado caminho no programa ou verificar um determinado requisito. Uma série de casos de teste é denominada conjunto de teste.

As atividades de testes estão associadas às atividades de desenvolvimento conforme podemos observar na Figura 2.1. Cada atividade de teste é destinada a identificar os tipos de defeitos que são criados na fase do processo de desenvolvimento ao qual ele está relacionado. As atividades de teste também são concebidas para identificar defeitos que não foram detectados por outras atividades de teste no início do processo de teste (McGregor, 2001).

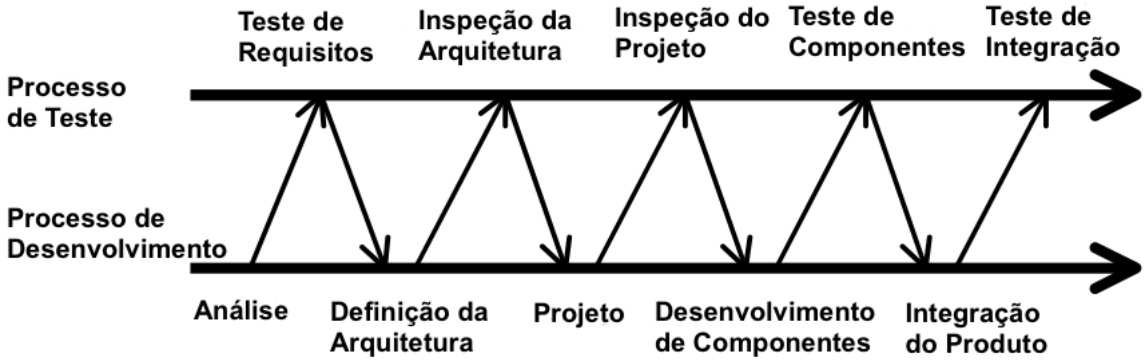

Figura 2.1: Teste e processo de desenvolvimento (McGregor, 2001).

A atividade de teste pode ser dividida em 4 fases distintas que são: teste de unidade, teste de integração, teste de sistemas e teste de regressão (Delamaro et al., 2007). O teste de unidade tem como foco as menores unidades de um programa, que são as funções, procedimentos, métodos ou classes. Desta forma, o teste pode ser aplicado à medida que ocorre a implementação, pelo próprio desenvolvedor. O teste de integração é realizado após serem testadas as unidades; tem como principal objetivo verificar se a interação entre as unidades está funcionando de maneira adequada, de forma que normalmente é executado pela própria equipe de desenvolvimento. Na fase de teste de sistema, o objetivo é verificar se as funcionalidades especificadas nos documentos de requisitos estão corretamente implementadas; estes testes muitas vezes podem ser realizados por equipes independentes. Por último, o teste de regressão é realizado durante a manutenção do software, de forma que a cada modificação efetuada, sejam realizados testes que mostrem se elas estão corretas e não adicionaram novos defeitos ao software.

Podemos modelar um programa P usando uma especificação S, podendo ser formal ou não, de forma que o programa P possa produzir saídas baseadas nas entradas de um domínio D. Se a saída produzida para todo elemento de D está de acordo com S, o programa está correto, ou seja, $\forall x \in D, S(x) = P(x)$. Uma falha ocorre quando uma saída não foi definida em S, indicando que P está em estado de erro. Um caso de teste é definido como um par ordenado $(x, S(x))|x \in D$, sendo que x é uma entrada definida por D, junto com uma saída, especificada por S, produzida por P ao receber x.

2.3 Critérios de Teste

Segundo o padrão IEEE 610.12 (IEEE, 1990), critérios que um sistema ou componente devem atender a fim de passar um determinado teste, são chamados de *critérios de teste*. Devido a dificuldade de se selecionar um número viável de elementos para $T \subseteq D$ (T é um subconjunto do domínio de testes D) para compor os casos de teste, foram elaboradas estratégias de seleção de casos de testes para melhorar a detecção de erros. Estes critérios devem auxiliar na atividade de teste respondendo a duas questões: quais elementos de D devem ser selecionados para compor os casos de teste e o quão bom é um subconjunto T para testar P (confiabilidade) (MALDONADO et al., 2004).

Técnicas de teste são classificadas de acordo com a origem da informação que é utilizada para estabelecer os requisitos de testes (Maldonado, 1991). As principais técnicas de teste são:

Funcional ou caixa-preta considera o sistema como uma caixa fechada onde não se tem conhecimento sobre sua implementação ou seu comportamento interno. Os testes são gerados somente considerando os valores de entrada e saída do sistema utilizando como base a sua especificação.

Estrutural ou caixa-branca estabelece os requisitos do software baseados na sua implementação. A geração dos testes considera as estruturas lógicas e funcionais implementadas, verificando se as funcionalidades e resultados gerados estão de acordo com a especificação, desta forma, o testador deve ter acesso ao código fonte do programa, que é utilizado para gerar os casos de teste.

Baseada em Defeitos estabelece os requisitos de teste explorando os defeitos típicos e comuns cometidos durante o desenvolvimento de software (Demillo, 1980). Várias características do desenvolvimento de software devem ser consideradas quando se trata do teste baseado em defeitos, como a linguagem utilizada, ferramentas, tipo de software, entre outros.

Neste trabalho são considerados os critérios de teste funcional, sendo que estes fazem uso das informações disponíveis em modelos para decidir quais casos de testes são mais relevantes. A seguir será apresentado em mais detalhes o tópico de teste baseado em modelos.

2.4 Teste Baseados em Modelos

A utilização de métodos precisos de desenvolvimento assegura um menor número de defeitos presentes no resultado final, pois resulta na redução de inconsistências e ambigüidades (Barroca e Mcdermid, 1992). Desta forma, podemos utilizar métodos formais para auxiliar no desenvolvimento de software, garantindo maior qualidade. Existem diversos modelos formais o quais podemos citar: MEFs (Gill, 1962), Statecharts (Harel, 1987), Redes de Petri (Petri, 1962), entre outras.

Máquinas de Estados Finitos (MEFs) são modelos utilizados para representar o comportamento de um sistema por um número finito de estados e transições (Gill, 1962). Modelos comportamentais são desenvolvidos em fases iniciais em um ciclo de desenvolvimento, permitindo o início de atividades de teste antes da fase de codificação (Apfelbaum e Doyle, 1997).

De acordo com Petrenko e Yevtushenko (2005) uma MEF é uma máquina de Mealy determinística podendo ser representada por uma tupla $M = (S, s_0, X, Y, D_M, \delta, \lambda)$, onde:

- S: é um conjunto de estados, incluindo o estado s_0 chamado de estado inicial,
- X: é um conjunto de símbolos de entrada,
- Y : é um conjunto de símbolos de saída,
- $D_M \subseteq S \times X$: é o domínio da especificação,
- $\delta : D_M \rightarrow S$ é uma função de transição e
- $\lambda : D_M \rightarrow Y$ é uma função de saída.

A MEF apresentada na Figura 2.2 também pode ser representada pela notação da tupla: $M = (\{s_0, s_1, s_2\}, s_0, \{a, b\}, \{0, 1\}, D_M, \delta, \lambda)$, contendo 3 estados, 2 entradas, 2 saídas, 6 transições, $D_M = \{(s_0, a), (s_0, b), (s_1, a), (s_1, b), (s_2, a), (s_2, b)\}$ e as funções de saída e transição representadas pela Tabela 2.1.

A MEF tem um estado ativado e em resposta a um evento de entrada pode ocorrer a mudança de estado e a produção de uma ação de saída. O novo estado e a saída produzida são definidos unicamente em função do estado ativado e da entrada. Portanto uma transição

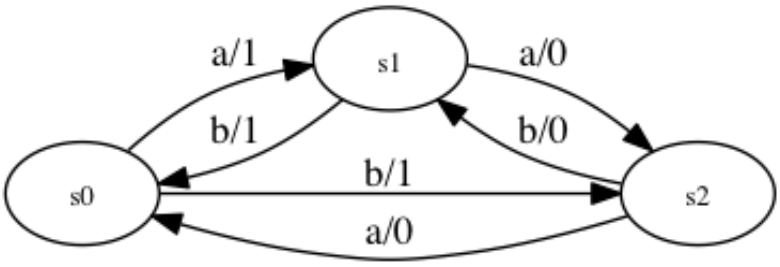

Figura 2.2: Exemplo de Máquina de Estado Finito

Tabela 2.1: Tabela de transição de estados

Estado atual	Entrada	Próximo estado	Saída
s_0	a	s_1	1
s_0	b	s_2	1
s_1	a	s_2	0
s_1	b	s_0	1
s_2	a	s_0	0
s_2	b	s_1	0

de estados é representada pelo par estado origem e entrada e pelo par saída e estado destino.

MEFs podem ser classificadas de diversas maneiras. Uma MEF é completamente especificada (ou completa) quanto existem transições definidas para todos os símbolos de entrada em cada estado da MEF. Caso contrário, a MEF é parcialmente especificada (ou parcial). A mesma também pode ser fortemente conexa quando para cada par de estados $s_i, s_j \in S$ existe uma seqüência que leva a MEF M do estado s_i ao estado s_j. Uma MEF é inicialmente conexa quando todos os estados são alcançáveis a partir do estado inicial. Por outro lado, a MEF é determinística se em cada estado, há uma única transição para cada entrada definida. Caso contrário, a MEF é não-determinística. Uma MEF também pode ser minimal quando não possui estados equivalentes.

A notação $\Omega(s)$ é usada para representar todas seqüências de entrada definidas para o estado s e Ω_M (abreviação de $\Omega(s_0)$) representa todas seqüências definidas para a MEF M. Uma seqüência vazia é representada pelo símbolo ϵ.

O conjunto de todas as MEFs determinísticas com o mesmo alfabeto de entrada de M de forma que todas as seqüências em Ω_M são definidas, é representada pela notação $\Im$. Para cada $N \in \Im$ podemos afirmar que $\Omega_M \subseteq \Omega_N$. O conjunto $\Im$ é chamado de domínio de defeitos para M (Simão e Petrenko, 2010).

Extensões comuns de duas seqüências são as seqüências obtidas por adicionar uma seqüên-

cia comum a elas.

Dois estados $s_i, s_j \in S$ são distinguíveis, se existe uma seqüência $\gamma \in \Omega(s_i) \cap \Omega(s_j)$, dado que $\lambda(s_i, \gamma) = \lambda(sj, \gamma)$. Um conjunto de testes é T-convergente (T-divergente) se cada par de seus testes são convergentes (divergentes) na MEF T.

Dado um conjunto não vazio de MEFs $\Sigma \subseteq \Im$ e dois testes $\alpha, \beta \in \Omega_M$, dizemos que α e β são Σ-convergente se eles convergem em cada MEF do conjunto Σ. Da mesma forma, dizemos que α e β são Σ-divergente se eles divergem em cada MEF do conjunto Σ.

Dois testes α e β em um dado conjunto de teste T são T-separáveis se existe uma extensão comum $\alpha\gamma, \beta\gamma \in T$, tal que $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$.

2.4.1 Métodos de geração e critérios de cobertura

Nesta seção são apresentados métodos para a geração de seqüências de teste baseada em especificação por MEFs. De acordo com Fujiwara et al. (1991) esses métodos devem gerar conjuntos de seqüências capazes de revelar os defeitos presentes em implementações, sem perder de vista que a quantidade de seqüências nos conjuntos não deve inviabilizar sua aplicação.

Seja S uma MEF representando uma especificação e I representando a implementação. Para validar a MEF I, deve garantir que esta seja equivalente a MEF S. Não é possível compará-las explicitamente, pois a implementação não é conhecida. Porém, é possível entrar com seqüências em I e obter suas saídas.

Basicamente, a estratégia consiste em gerar seqüências de entradas, a partir de uma MEF conhecida, cujas saídas só poderiam ser reproduzidas por uma outra MEF equivalente. Dessa forma, aplicando as seqüências geradas a partir da especificação S na implementação I e comparando as saídas obtidas, é possível garantir que a implementação representa o mesmo comportamento da especificação.

Caso S e I possuam o mesmo número de estados n, o conjunto de testes gerado é chamada de *n-completo* e este capaz de revelar todos as falhas da implementação.

Muitos métodos de geração de seqüências de teste utilizam dois conceitos: o *State Cover* e o *Transition Cover*. *State cover* é um conjunto de seqüências de entradas que faz com que todos os estados de uma MEF sejam atingidos. O *state cover* é representado pelo conjunto

Q, contendo n seqüências, sendo n o número de estados, incluindo a sequência vazia ϵ. O *Transition cover* é um conjunto de seqüências de entradas capaz de atingir pelo menos uma vez cada uma das transições de uma MEF. O conjunto *transition cover* é representado por P e este inclui o conjunto *state cover*, pois é necessário passar por todos os estados para atingir todas as transições.

O tamanho de um conjunto de testes $\omega(T)$ é calculado pela soma dos comprimentos das seqüências contidas em T que não são prefixos próprios de outras seqüências. Faz parte desta seqüência a operação *reset*, representada como "r". Esta é adicionada no início dos casos de teste, e "reinicia" corretamente a MEF, ou seja, leva a MEF ao seu estado inicial.

Existem diversos métodos de teste como por exemplo: Método W (Chow, 1978), Método HSI (Luo et al., 1994) e o Método P (Simão e Petrenko, 2010).

O método W, anteriormente chamado de *Automata Theoretic*, utiliza um conjunto de caracterização que consiste de seqüências de entrada que, através das saídas correspondentes, pode distinguir cada um dos estados de uma MEF mínima.

Método HSI

O método HSI é uma variação do método W cuja finalidade é reduzir o tamanho do conjunto de teste gerado. A principal melhoria deste método é que ele é aplicável em MEFs parciais, enquanto os outros métodos, em geral, necessitam uma MEF completa. Para cada estado do método HSI, um conjunto H_i é construído para distinguir o estado s_i dos demais. A construção do conjunto H_i é feita pela união das seqüências de separação de cada par de estados da MEF.

A geração dos casos de testes utilizando o método HSI é dividido em duas etapas. A primeira etapa ocorre a geração das seqüências por $r.Q \otimes HSI$, onde $Q \otimes HSI = \bigcup_{q \in Q} \{q\}.H_i$. Em seguida, são geradas seqüências por $r.P \otimes HSI$, onde $P \otimes HSI = \bigcup_{p \in P} \{p\}.H_i$. Desta forma, i é definido pelo estado final da execução de q, na primeira, e p na segunda etapa.

Com base na MEF da Figura 2.2 temos: $Q = \{\epsilon, a, b\}$, $P = \{\epsilon, a, aa, ab, b, ba, bb\}$ e os conjuntos H_i são: $H_0 = \{a\}$, $H_1 = \{ab\}$ e $H_2 = \{ab\}$.

Na primeira etapa, as seqüências geradas são representadas por: $\{\epsilon H_0, aH_1, bH_2\}$ resultando em $\{a, aab, bab\}$. Na segunda etapa, as seqüências geradas são representadas por: $\{\epsilon H_0, aH_1, aaH_2, abH_0, bH_2, baH_0, bbH_1\}$ resultando em $\{a, aab, aaab, aba, bab, baa, bbab\}$.

Para obter o conjunto de testes HSI devemos remover os prefixos e acrescentar a função reset (r), gerando o conjunto $TS_{HSI} = \{raaa, raab, raba, rbaa, rbba, rbbb\}$ de tamanho 24.

Método P

O método P, diferente de métodos que garantem a cobertura de falhas n-completo, permite a geração de casos de teste p-completo para $p < n$. Desta forma, os casos de teste n-completo também são p-completo para qualquer $p \leq n$. O método P permite a extensão de casos de teste definido pelo usuário, podendo conter apenas uma sequência vazia, até a mesma se tornar p-completo, para uma MEF $p \leq n$.

O algoritmo para geração de conjuntos p-completo são baseados no Teorema 1 e Lemas de 1 a 3, sendo estes apresentados a seguir.

Teorema 1: Dado T ser um conjunto de testes para uma MEF M com n estados e $p \leq n$. Temos que T é um conjunto de testes p-completo para M se:

(i) $p < n$ e T contém um conjunto $\Im(T)$-divergente com $p + 1$ testes; ou

(ii) $p = n$ e T contém um conjunto $\Im(T)$-*convergence-preserving*, conjunto de cobertura de transições inicializado para M.

Lema 1. Dado um conjunto não vazio Σ de MEF determinística reduzida com o mesmo alfabeto de entrada, as seguintes propriedades são válidas:

(i) Extensões comuns de testes Σ-convergente também são Σ-convergentes.

(ii) Testes que possuam extensões comuns Σ-divergentes também são Σ-divergentes.

(iii) Dado dois testes Σ-divergentes, qualquer teste Σ-convergente com um deles é Σ-divergente com o outro.

(iv) Se testes $\alpha\varphi^k$ são Σ-divergentes, para $K > 1$, então α e $\alpha\varphi$ também são Σ-divergentes.

(v) Se testes α e $\alpha\beta\gamma$ são Σ-convergentes e testes α e $\alpha\gamma$ são Σ-divergente, então $\alpha e \beta$ também são Σ-divergentes.

Lema 2. Dado um conjunto de teste T de uma MEF M, testes T-separáveis são $\Im(T)$-divergentes.

Lema 3. Dada uma suite de teste T e $\alpha \in T$, temos K é um conjunto $\Im_n(T)$-divergente com n testes e $\beta \in K$ é um teste M-convergente com α. Se α é $\Im_n(T)$-divergente com cada teste em $K \backslash \{\beta\}$, então α e β são $\Im_n(T)$-convergente.

Dada a MEF M, um conjunto de teste T e $\alpha \leq n$, o algoritmo gera os conjuntos de teste que contém T e satisfaçam as condições do Teorema 1 e resultando o conjunto de teste p-completo.

Os conjuntos $\Im_n(T)$-convergente e $\Im_n(T)$divergente são atualizados pela aplicação de um conjunto de regras que inferem novas relações a partir das relações existentes, conforme Lema 1. A seguir são apresentadas essas regras:

- Regra 1: Se (α, β) é adicionado a C, para cada $(\alpha, \chi) \in C$, adicionamos (β, χ) em C.
- Regra 2: Se $(\alpha, beta)$ é adicionado a C, então, para todas suas extensões comuns $\alpha\varphi, \beta\varphi \in T$, adiciona $(\alpha\varphi, \beta\varphi)$ em C (Lema 1(i)).
- Regra 3: Se (α, β) é adicionado a D, e eles são extensões comuns de testes α' e β', então adiciona (α', β') a D (Lema 1(ii)).
- Regra 4: Se (α, β) é adicionado em C, então, para cada $\chi \in T$ se $(\alpha, \beta) \in D$, adiciona (β, χ) em D; Se $(\beta, \chi) \in D$, adicione (α, χ) a D (Lema 1(iii)).
- Regra 5: Se (α, β) é adicionado em D, então, para cada $\chi \in T$, se $(\alpha, \chi) \in C$, adiciona (β, χ) em D; Se $(\beta, \chi) \in C$, adicione (α, χ) em D (Lema 1(iii)).
- Regra 6: Se $(\alpha, beta)$, com $\alpha \leq \beta$, é adicionado em D e existe seqüências φ e $k > 1$, tal que $\beta = \alpha\varphi^k$, então adicione $(\alpha, \alpha\varphi)$ em D (Lema 1(iv)).
- Regra 7: Se $(\alpha, \alpha\beta, \gamma)$ é adicionado em C, e $(\alpha, \alpha\gamma) \in D$, então adicione (α, α, β) em D (Lema 1(v)).
- Regra 8: Se (α, α, γ) é adicionado a D então, para cada sequência β, tal que $(\alpha, \alpha, \beta, \gamma) \in C$, adicione (α, α, β) em D (Lema 1(v)).
- Regra 9: Se (α, α, γ) é adicionado em C, então, para cada sequência β tal que $(\beta, \beta gamma) \in D$, adicione (α, β) em D (Lema 1(vi)).
- Regra 10: Se $(\beta, \beta\gamma)$ é adicionado em D, então para cada sequência α tal que $(\alpha, \alpha\gamma) \in C$, adicione (α, β) em D (Lema 1(vi)).

Com base na MEF ilustrada pela Figura 2.2, foi gerado o conjunto $TS_P = \{rbaa, raaaa, raba, rbbaa\}$ de tamanho 18.

Outras representações das seqüências de teste

As seqüências de teste, podem ser representadas pela *árvore de teste*, formada por nós e arestas, de maneira que cada nó da árvore representa uma sequência formada pelos símbolos de entrada do nó raiz até o atual, e as arestas representam a transição da MEF, rotulada pela entrada e saída produzida. Baseada na MEF da Figura 2.2, foi elaborado a árvore de teste que utiliza o conjunto de testes $TS_{HSI} = \{\ raaa, raab, raba, rbaa, rbba, rbbb\}$ (Figura 2.3).

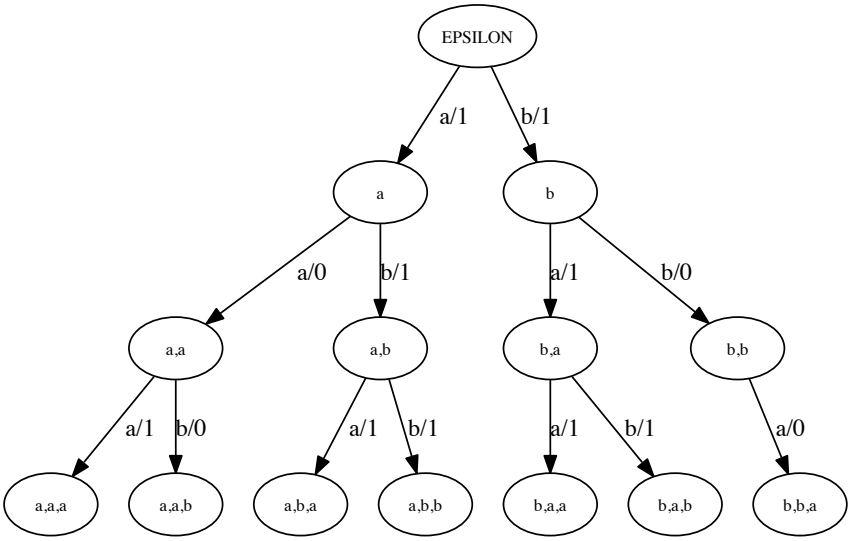

Figura 2.3: Árvore de teste considerando a MEF da Figura 2.2

Outra forma de representar o conjunto de seqüências de teste, é utilizar um grafo de distinção, cada nó é representado pela sequência e pelo estado atingido com esta sequência. A principal diferença é que os nós do grafo apenas são ligados se as seqüências são T-distinguíveis. A Figura 2.4 apresenta o grafo de distinção considerando a MEF e árvore de testes vistas anteriormente, Figura 2.2 e 2.3 respectivamente.

Um *clique* de um grafo é um conjunto de vértices, tal que cada vértice possui ligação com os demais vértices desse mesmo conjunto. As arestas destacadas no grafo representam um clique de tamanho *3*.

2.5 Considerações Finais

Neste capítulo foram apresentados os principais conceitos que envolvem teste de software, bem como técnicas e critérios de teste. Vimos também como funciona a realização de testes

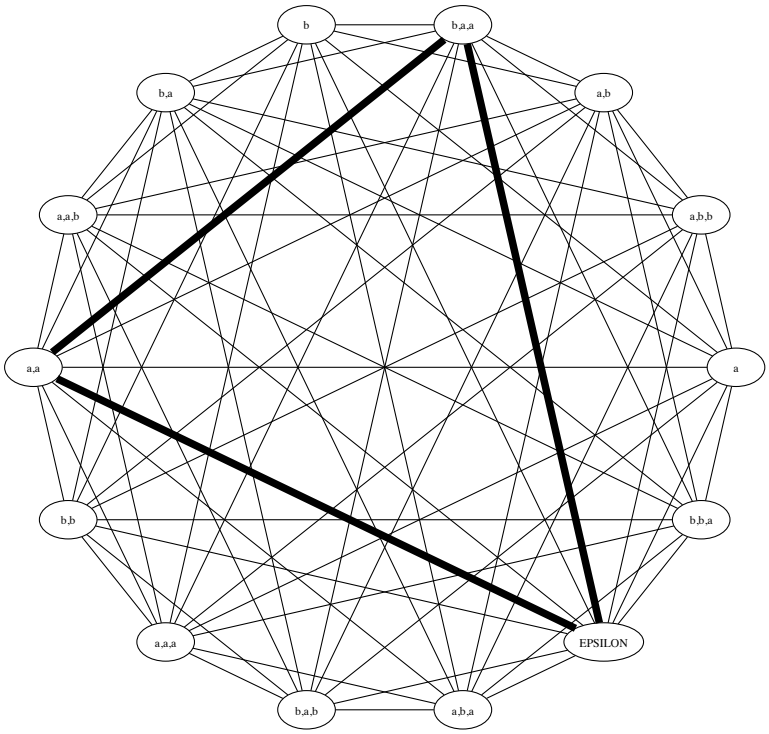

Figura 2.4: Grafo de Distinção considerando a MEF da Figura 2.2 e árvore de teste Figura 2.3

baseados em modelos utilizando MEFs. Por último, discutimos alguns métodos de geração de casos de testes.

Os conceitos de MEFs e os métodos de geração de casos de testes forneceram o embasamento necessário para a elaboração da estratégia de testes, sendo cruciais para a realização e reutilização dos testes entre os produtos da LP, conforme será apresentado no Capítulo 4. No próximo Capítulo, veremos os principais conceitos e abordagens de testes para LP.

Capítulo

3

Linha de Produto de Software

3.1 Considerações Iniciais

No Capítulo anterior, discutimos sobre os principais conceitos de testes, métodos e técnicas o qual, fornecem o embasamento teórico necessário para os próximos Capítulos.

Neste Capítulo, serão abordados conceitos e definições sobre Linha de Produto de Software bem como suas principais características. Na Seção 3.3 é apresentado o conceito de variabilidade e sua classificação. Logo em seguida, na Seção 3.4, é apresentado outro conceito importante de LP que é o modelo de características. Na Seção 3.5, são apresentadas algumas abordagens de criação de LPs. O conceito de Linguagens de Domínio Específico é discutido na Seção 3.6. Os conceitos de Testes em LP e as principais abordagens de teste em LP são discutidos na Seção 3.7. Por fim, na Seção 3.8 é apresentado as considerações finais deste capítulo.

3.2 Conceitos e Definições

LP é uma abordagem que visa a construção sistemática de software baseada em uma família de produtos por meio da reutilização de artefatos (Gimenes e Travassos, 2002). Uma família de produtos de software é um conjunto de produtos de software com propriedades suficientemente similares. Essas propriedades permitem a definição de uma infraestrutura comum dos itens que a compõem e a parametrização das diferenças entre eles, normalmente conhecida como núcleo de artefatos.

As atividades essenciais de uma abordagem de LP, conforme mostra a Figura 3.1, são: Desenvolvimento do Núcleo de Artefatos (Engenharia de Domínio), Desenvolvimento do

Produto (Engenharia de Aplicação), e o Gerenciamento da Linha de Produto de Software (SEI, 2011).

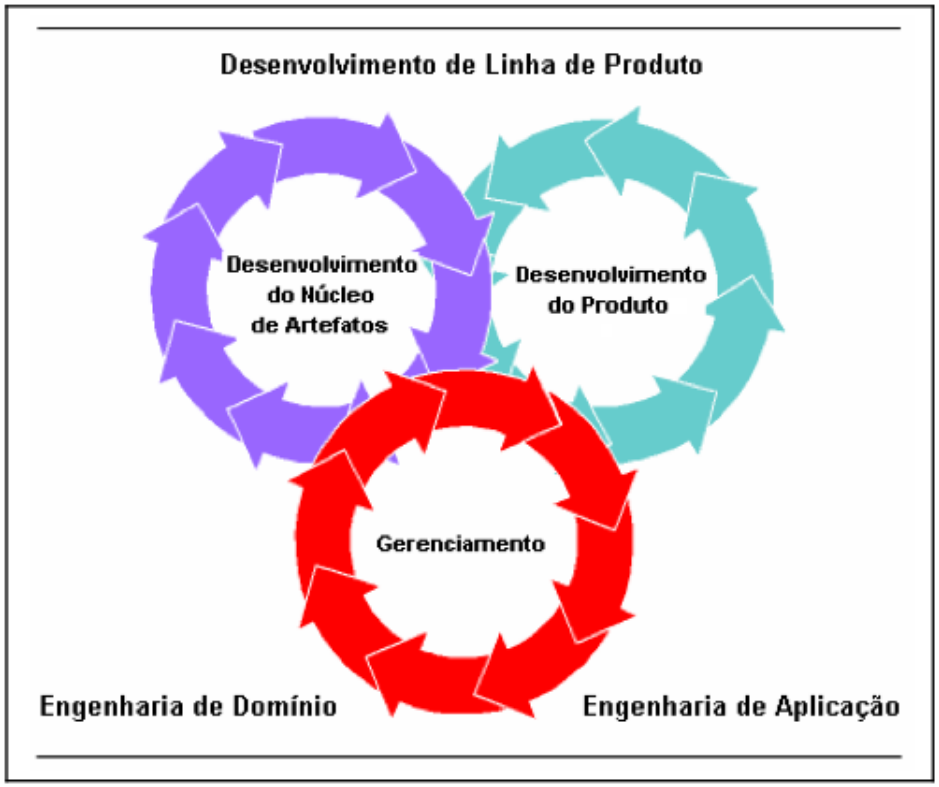

Figura 3.1: Principais atividades da engenharia de linha de produto de software (SEI, 2011)

A atividade Engenharia de Domínio é responsável por desenvolver o núcleo de artefatos da LP o qual estabelece uma infraestrutura central que será reutilizada pelos produtos gerados partir da LP. A atividade Engenharia de Aplicação é responsável por gerar produtos de uma LP, por meio da criação de uma instância da arquitetura central. Por último, Gerenciamento da Linha de Produto engloba as atividades relacionadas à construção e manutenção da LP, realizadas com base em planos que definem as estratégias a serem seguidas pela organização.

3.3 Variabilidade

Variabilidades representam as diferenças entre os produtos de uma LP (Heymans e Trigaux, 2003). Elas são descritas por pontos de variação e variantes.

Segundo van der Linden et al. (2007), quando se gerencia variabilidade em LP, é preciso distinguir três tipos principais:

- Variabilidade Comum: uma característica (funcional ou não funcional) pode ser comum a todos os produtos da linha de produto;
- Variabilidade: uma característica pode ser comum a alguns produtos, mas não a todos. Deve então ser explicitamente modelada como uma possível variabilidade e deve ser

implementada de maneira que permita tê-la em produtos selecionados;

- Produto específico: uma característica pode ser parte de apenas um produto.

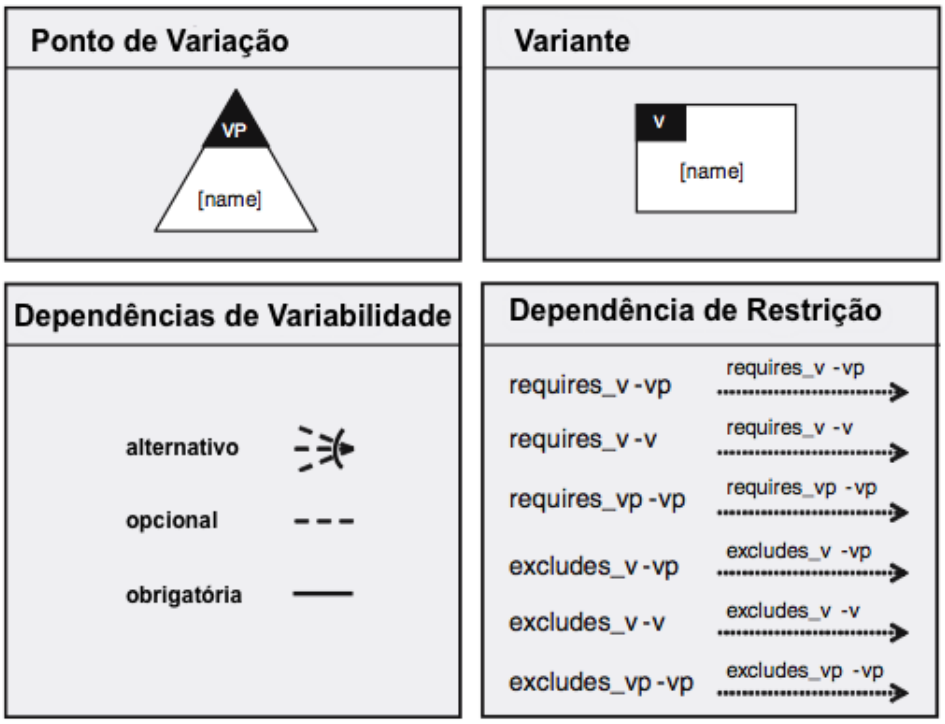

Figura 3.2: Representação gráfica da variabilidade (van der Linden et al., 2007).

Existem várias formas de representar variabilidades, uma delas é utilizar uma notação gráfica conforme mostrado na Figura 3.2. Os elementos principais são (van der Linden et al., 2007):

- *Ponto de variação*: o ponto de variação descreve onde existem diferenças nos sistemas finais;
- *Variantes*: são as diferentes possibilidades que existem para satisfazer um ponto de variação;
- *Dependências de variabilidade*: este é utilizado como base para designar as diferentes escolhas (variantes) que são possíveis para preencher um ponto de variação. A notação inclui uma cardinalidade que determina quantas variantes podem ser selecionadas simultaneamente;
- *Dependências de restrição*: dependências entre elas descrevem certas variantes selecionadas. Existem duas formas:
  - Requerida: a escolha de uma variante específica pode exigir a escolha de outra variante;

– Excluída: a escolha de uma variante específica pode proibir a escolha de outra variante.

3.4 Modelo de Características

O modelo de características é utilizado para representar as variabilidades e similaridades entre os produtos de uma LP. Um modelo de características é uma representação de particularidades relevantes de alguma entidade de interesse. Uma característica pode ser definida como uma propriedade de um sistema que é relevante para alguma entidade envolvida em seu desenvolvimento ou uso. Ela é usada para capturar pontos comuns ou estabelecer uma discriminação entre sistemas em uma família de sistemas. As características podem denotar qualquer propriedade funcional ou não funcional nos níveis de requisitos, de arquitetura, de componentes ou de plataformas computacionais (Czarnecki et al., 2005).

Modelos de características são normalmente organizados em diagramas hierárquicos, na forma de árvore, em que cada nó representa uma característica e cada característica pode ser descrita por um conjunto de sub-características representadas como nós descendentes (Cechticky et al., 2004).

Figura 3.3: Exemplo modelo de características (van der Linden et al., 2007).

A Figura 3.3 representa um modelo de características da segurança de uma casa, representando o ponto de variação "segurança da casa" contendo duas possíveis variantes "cámera de vigilância" e "detecção de movimento".

3.5 Abordagens de Linha de Produto de Software

Existem diversas abordagens de LP dentre as quais pode-se citar algumas, como FODA (Feature-Oriented Domain Analysis) (Kang et al., 1990), FAST (Family-Oriented Abstraction, Specification and Translation) (Weiss e Lai, 1999), Product Line Software Engineering (PuLSE) (Bayer et al., 1999), e mais recentemente, a abordagem Product Line UML-Based Software Engineering (PLUS) (Gomaa, 2005).

O método FODA (Kang et al., 1990) é um dos precursores da abordagem de LP. Seu foco principal é a identificação das características do sistema no domínio. Este método oferece a descrição do produto no domínio de análise, bem como os processos gerados por meio deste. O método FODA visa o desenvolvimento do domínio de produtos que são genéricos e amplamente aplicáveis dentro de um domínio.

O método PLUS (Gomaa, 2005) oferece um conjunto de conceitos e técnicas para estender métodos de projeto baseados em UML e no processo unificado. O PLUS possui as seguintes fases: modelagem de requisitos em LP, modelagem de análise em LP, modelagem de projeto em LP e engenharia de aplicação de software.

3.6 Linguagem Específica de Domínio

Linguagem Específica de Domínio é uma linguagem de programação altamente abstrata que oferece uma maneira natural e intuitiva de lidar com um domínio lógico específico. Elas podem servir como APIs fáceis e flexíveis para os programadores ou permitir que os clientes tenham controle sobre a forma como o sistema lida com a sua lógica de negócios (Goldstein, 2009).

van Deursen e Klint (1998) afirmam que os principais benefícios do uso de DSL são: permitir que soluções sejam expressas no idioma e no nível de abstração do domínio do problema. Especialistas podem entender, validar, e modificar o software, adaptando o Domínio de Descrições Específicas (DSDs); modificações são mais fáceis de fazer e seu impacto é mais fácil de entender; o conhecimento explicitamente disponível pode ser reutilizado em diferentes aplicações; e a forma como o conhecimento é representado é independente da plataforma de implementação.

Apesar dessas vantagens, Fowler (2010) apresenta alguns problemas na utilização de DSL, que incluem: alta curva de aprendizagem; custos elevados na criação da DSL; alto custo de educação dos usuários de DSL; dificuldade de concepção de uma DSL; e a dificuldade de migração dos scripts DSL.

O desenvolvimento de uma DSL tipicamente envolve três etapas distintas: análise, implementação e uso. Na etapa de análise as seguintes atividades são realizadas: identificar o domínio do problema; reunir os conhecimentos relevantes neste domínio; agrupar esses conhecimentos de ponta em um conjunto de noções semânticas e as operações sobre eles;

e projetar uma DSL que descreve de forma concisa aplicações no domínio. Na etapa de implementação ocorrem determinadas atividades: construir uma biblioteca que implementa as noções semânticas; e formular e implementar um compilador que realiza a tradução de programas DSL para uma seqüência de chamadas de biblioteca. Na última etapa do uso da DSL ocorrem as atividades de escrever programas DSL para todas as aplicações desejadas e compilá-los (van Deursen et al., 2000).

DSLs são importantes no contexto deste projeto pois no domínio de sistemas embarcados é comum o uso dessas linguagens. Neste projeto foi utilizado duas DSLs, para a especificação de MEFs, sendo vital para a viabilidade da estratégia de teste proposta.

3.7 Teste em Linha de Produto de Software

Teste em LP consiste na utilização de técnicas de teste de sistemas únicos (ex: teste unitário, teste de integração, teste de regressão). Porém, o processo de testes é dividido em duas etapas: teste do núcleo de artefatos de software (Engenharia de domínio) e testes dos produtos de software específico (Engenharia de aplicação). Na engenharia de domínio são testados os artefatos de domínio, as características comuns e características mais utilizadas. Na engenharia de aplicação são feitos os demais testes conforme as características selecionadas do produto.

Seis princípios essenciais para teste de sistema desenvolvidos em LP são apresentados a seguir (Pohl e Metzger, 2006):

- *Preservar a variabilidade nos Artefatos de Teste de Domínio*: testes de sistema são realizados para avaliar se um sistema está em conformidade com suas necessidades. Além dos requisitos de domínio, a variabilidade definida em uma LP deve ser levada em conta quando os artefatos de teste de sistema forem derivados para aplicações da LP. Para considerar a variabilidade no teste do sistema, pode ser definida explicitamente a variabilidade nos artefatos de teste de domínio e relacioná-la com a variabilidade definida nos requisitos de domínio.

- *Teste de semelhanças em Engenharia de Domínio*: um defeito descoberto em uma característica comum de uma LP afetará todos os produtos da LP e, portanto, afetará a qualidade geral da LP. Portanto, devem ser testados os aspectos comuns da LP tão cedo quanto possível, de preferência no processo de engenharia de domínio.

- *Utilizar aplicações de referência para determinar defeitos em variantes mais usadas*: se uma variante é utilizada na maioria dos produtos da LP, um defeito descoberto nesta variante pode afetar a qualidade LP tanto quanto um defeito em uma característica comum. Portanto, devem ser testadas todas as variantes que possam ser usadas em muitos produtos LP tão cedo quanto possível. Desta forma, deve ser utilizado um

processo de gerenciamento de variabilidades para identificar quais são as características mais utilizadas.

- *Teste de pontos comuns baseado em um produto de referência*: se um produto de referência é usado para testar variações freqüentemente usadas, este também pode ser usado para testar os pontos comuns da LP. Desse modo, o esforço adicional necessário para implementar placeholders podem ser reduzidos.

- *Teste preciso das ligações de variabilidades*: ao relacionar os pontos de variação dos artefatos de domínio para derivar produtos de uma LP, podem ocorrer erros no processo de instanciação. Por exemplo, um produto da LP poderá incluir variantes que não deveriam ser incluídas no produto. Da mesma forma, uma variante pode ser omitida, não sendo vinculada com o produto. Para descobrir se houve omissão de variantes podem ser utilizados os testes de sistemas que falhará na falta de funcionalidades. Para verificar se houve a inclusão indesejada de variantes devem ser definidos casos de teste adicionais.

- *Reutilização dos artefatos de teste de produto em diferentes produtos*: dois ou mais produtos de uma LP podem ser relacionados a um ou mais pontos de variação, como resultado, estes produtos conterão um conjunto comum de variantes. Neste caso, os casos de teste e os resultados obtidos por meio da execução dos testes, que consideram estas variantes podem ser reutilizados entre os produtos. Portanto, o esforço de teste pode ser significativamente reduzido.

A Figura 3.4 ilustra a relação entre LP e testes (McGregor, 2001). O plano de teste e os casos de testes são construídos em nível de LP e especializados para cada produto. Ao especializar o produto deve ser garantido que o seu comportamento seja o mesmo que o da LP.

Existem quatro diferentes estratégias para modelar teste em família de produtos e para integrar teste em engenharia de família de produtos, sendo (Tevanlinna et al., 2004):

- A estratégia mais facilmente implementada é confiar na engenharia de aplicação para entrega de produtos bem testados. Esta abordagem não é orientada a família de produtos, de maneira que o teste é realizado para cada produto instanciado da LP, sendo assim, são suavizados os benefícios da reutilização, mas com isto a qualidade dos produtos é mais facilmente garantida.

- Uma abordagem que explora as semelhanças dos produtos em teste incremental de famílias de produtos. Esta abordagem, o primeiro produto é testado individualmente e os produtos seguintes são testados usando técnicas de teste de regressão;

- Uma abordagem que permite a instanciação dos artefatos reutilizáveis, desta forma, os artefatos de teste são criados o mais amplamente possível na engenharia de domínio

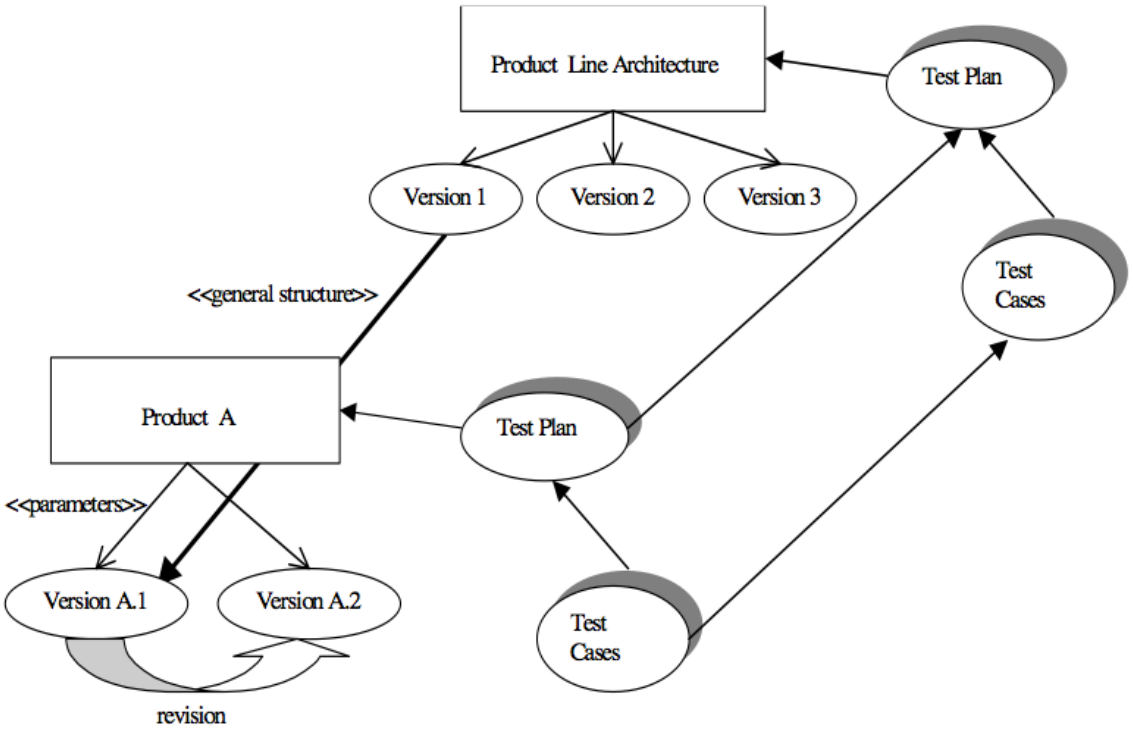

Figura 3.4: Relação entre LP e testes (McGregor, 2001).

antecipando também as variabilidades, criando, por exemplo, moldes de documentos e casos de teste abstratos (testes gerados a partir de um modelo abstrato).

- Divisão de responsabilidades, em que o teste em nível do modelo V é dividido entre as diferentes unidades de engenharia. Uma divisão, por exemplo, é ter testes unitários em componente comuns na engenharia de domínio. Quando uma unidade de engenharia de aplicação reúne um produto, é responsável pelos testes de sistema e aceitação.

Apesar da pesquisa em teste de LP ser recente, existem diversas abordagens na literatura dentre as quais pode-se citar: Reutilização de Testes baseados em Modelos (Olimpiew e Gomaa, 2009); Testes baseados em Caso de Uso de LP (Bertolino e Gnesi, 2003); O Modelo W para teste em LP (Jin-hua et al., 2008); Abordagem de Teste baseado em Especificação para LP (Uzuncaova et al., 2007); Teste em LP usando Geração de Teste Incremental (Uzuncaova et al., 2008); Automatizando a Definição de Caso de Teste usando Linguagem Específica de Domínio (Im et al., 2008). A seguir uma visão geral destas abordagens será apresentada:

Reutilização de Testes baseados em Modelos: este método de reutilização de teste baseado em modelos para LP é usado para criar especificação de testes a partir dos casos de uso e do modelo de características. Este método é utilizado para reduzir o número de especificação de testes reutilizáveis, criado para cobrir todos os cenário de casos de uso, todas as características, e combinações de características selecionadas da LP (Olimpiew e Gomaa, 2009).

Testes baseados em Caso de Uso de LP: O PLUTO (Product Line Use Case Test Optimi-

sation) é um método usado para gerenciar o processo de teste de LP, descrito como Casos de Uso de Linha de Produtos (PLUCs). PLUCs é uma extensão dos conhecidos Casos de Uso Cockburn, uma notação baseada em descrições de requisitos em linguagem natural (Cockburn, 2000). O método de teste proposto é baseado em partição de categorias e pode ser usado para derivar uma especificação de teste genérico para LPs, e um conjunto de cenários de teste relevantes para uma aplicação de um cliente específico (Bertolino e Gnesi, 2003).

O modelo W para teste em LP: O modelo W, descreve dois sub-modelos separados e intimamente relacionados que são: teste de domínio e teste de aplicação. O Modelo W complementa o ciclo de vida da LP, adicionando três atividades de teste na fase de engenharia de domínio, sendo: teste de componentes, teste de integração e teste de plataforma. Da mesma forma, na fase de engenharia de aplicação também são adicionados três atividades de testes que são: teste de sistema, teste de integração e testes unitários. Dessa forma, é garantida a qualidade dos ativos comuns e reutilização correta dos mesmos, validando diferentes artefatos o mais cedo possível (Jin-hua et al., 2008).

Abordagem de Teste baseado em Especificação para LP: utiliza especificações dadas como fórmulas em Alloy, uma linguagem de modelagem estrutural baseada em lógica de primeira ordem (MIT, 1997). Fórmulas Alloy podem ser analisadas satisfatoriamente utilizando o Alloy Analyzer. O analisador traduz as fórmulas Alloy para uma fórmula proposicional e encontra uma instância usando a tecnologia SAT3. Cada programa em uma LP é especificado como uma composição de características, onde cada característica representa uma fórmula Alloy. Testes são gerados para resolver a fórmula resultante (Uzuncaova et al., 2007).

Teste em LP usando Geração de Teste Incremental: é uma abordagem baseada em especificação para a geração de testes para os produtos em uma LP. Dadas as características como fórmulas de lógica de primeira ordem, esta abordagem utiliza análise baseada em SAT para gerar automaticamente entradas de teste para cada produto em uma LP. Para garantir a solidez de geração, utiliza uma técnica automática para o mapeamento de uma fórmula que especifica uma característica em uma transformação que define o refinamento incremental de suítes de teste (Uzuncaova et al., 2008).

Automatizando a Definição de Caso de Teste usando Linguagem Específica de Domínio: Esta abordagem é utilizada para automatizar a definição de caso de teste no contexto de aplicar a abordagem dirigida a modelos para o desenvolvimento de uma LP. Dessa forma, os casos de teste são automaticamente extraídos de casos de uso especificados usando uma DSL. A estrutura da DSL fornece padrões de projeto de teste que são pistas necessárias para extrair automaticamente os casos de teste. Um conjunto de ferramentas orientado a modelos é utilizado para automatizar o processo de teste do sistema, que começa com um modelo de caso de uso e termina com a execução automática de testes do sistema (Im et al., 2008).

3.8 Considerações Finais

Neste capítulo, discutimos sobre os principais conceitos de LP, como variabilidade, modelo de características, as principais abordagens para a criação de LPs, e DSL. Vimos também conceitos e estratégias de teste para LPs, bem como diversas abordagens de teste para LP.

As abordagens de teste discutidas neste trabalho bem como o capítulo de Teste de Software e os conceitos de LP serviram como embasamento teórico para o entendimento e criação da estratégia de teste incremental para LP, que será apresentada no próximo Capítulo.

Capítulo

4

FSM-TSPL - Teste Incremental Baseado em Máquina de Estados Finitos para LP

4.1 Considerações Iniciais

No Capítulo 2 vimos alguns métodos de geração de casos de testes baseados em MEF, dentre os quais utilizamos o método HSI (Luo et al., 1994) e P (Simão e Petrenko, 2010). A estratégia FSM-TSPL visa testar produtos instanciados de uma LP, de maneira que possa reutilizar os conjuntos de teste entre os produtos da LP. Para garantir a conformidade entre a especificação do produto e a implementação é utilizado o método P, o qual foi escolhido devido ao mesmo permitir que seja informado um conjunto de testes inicial que pode ser incrementado até atingir a cobertura desejada e também por este gerar, na grande maioria das vezes, conjunto de testes menores que outras abordagens. Estas características o tornam único e foram primordiais para ter uma boa integração com a LP, permitindo a reutilização de testes entre os produtos da mesma.

Como apresentado anteriormente no Capítulo 3, diversas abordagens de teste foram utilizadas como base para a compreensão dos princípios de testes de LP. Apesar dessas abordagens possuírem características comuns, existe uma diferença na forma de reutilizar os testes, e resultado dos mesmos, entre os produtos instanciados da LP. Desta forma, neste capítulo será apresentada a estratégia FSM-TSPL que propõe a reutilização de testes entre os produtos instanciados de uma LP baseado em MEFs, visto que a reutilização de testes entre produtos é um dos seis princípios de teste em LP (Pohl e Metzger, 2006), apresentado no capítulo

anterior. Neste contexto, a *reutilização de testes* ocorre por meio da reutilização de conjuntos de teste entre os produtos instanciados pela LP.

De maneira a facilitar o entendimento da estratégia FSM-TSPL será utilizado a LP AGM. A AGM (SEI, 2011) é uma LP pedagógica que produz alguns jogos arcades como Brickles, Pong e Bowling. Foi criada pelo *Software Engineering Institute* (SEI), contém um conjunto de documentos e modelos UML que facilitam o aprendizado dos conceitos de LP. Os jogos são controlados por um jogador, tendo como objetivo obter mais pontos acertando os obstáculos mostrados na tela.

Os principais artefatos da AGM são: modelo de características, o modelo de casos de uso, o modelo de classes e a arquitetura lógica dos componentes (de Oliveira Júnior, 2010). A seguir será apresentado a descrição e explicação do modelo de características, sendo este composto por quatro características principais, representado pela Figura 4.1:

- *Serviços*, define os serviços *jogar*, *pausar* e *salvar* para cada jogo;
- *Regras*, define as regras a serem seguidas em cada jogo;
- *Configuração*, define as configurações básicas para cada jogo;
- *Ação*, define as ações durante o jogo (ex. movimentação, colisão).

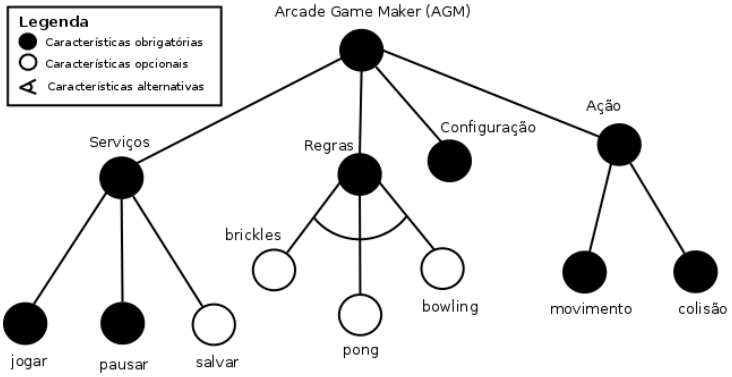

Figura 4.1: Modelo de Características AGM adaptato (SEI, 2011).

Baseado no modelo de características da LP AGM, foi modelado um diagrama de estado o qual representa o funcionamento geral da LP, conforme Figura 4.2.

4.2 FSM-TSPL

A FSM-TSPL começa após a instanciação de produtos da LP e das mesmas serem criadas usando qualquer abordagem de desenvolvimento LP (ex. FODA, PLUS ou PuLSE), como

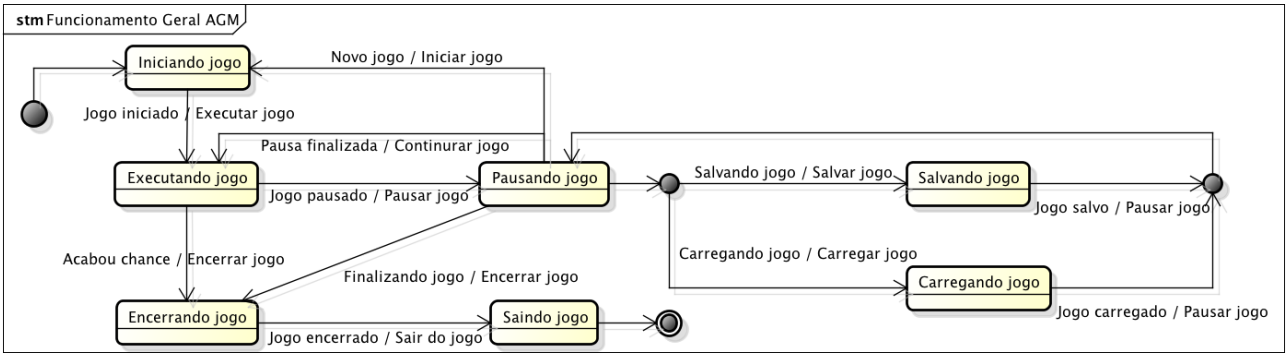

Figura 4.2: Diagrama de Estados - Funcionamento Geral da LP AGM

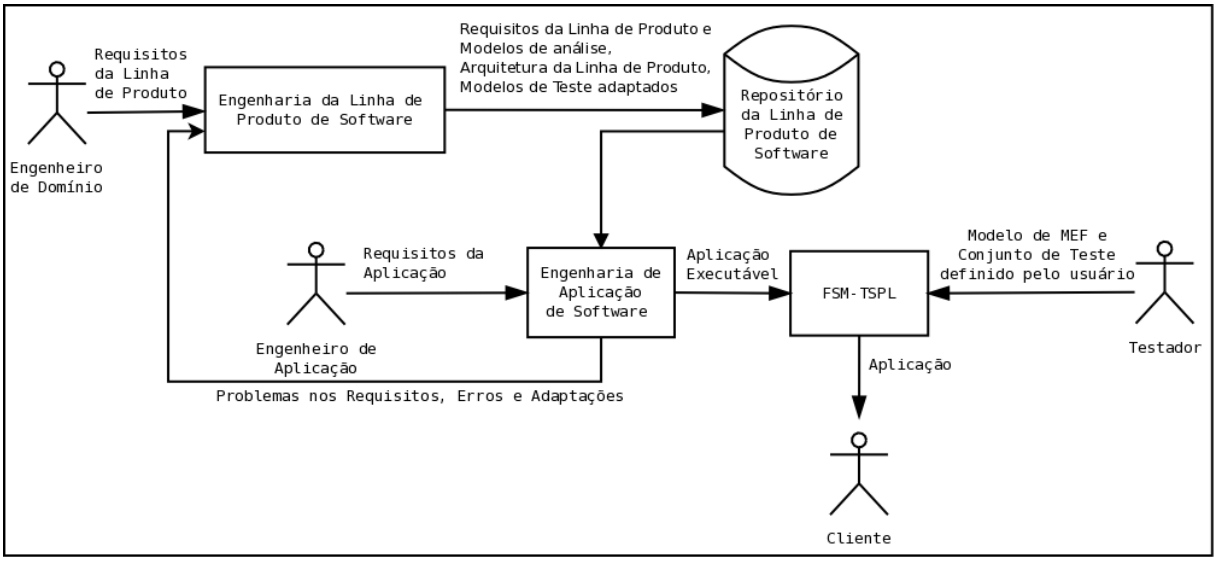

Figura 4.3: Processo de geração de casos de teste da estratégia FSM-TSPL adaptado de (Olimpiew e Gomaa, 2009).

mostrado na Figura 4.3.

O Engenheiro de Domínio fornece os requisitos necessários para o início da fase de Engenharia da Linha de Produto (Engenharia de Domínio). Nesta fase são gerados os Modelos de análise e Arquitetura da LP, que em conjunto com os Requisitos da LP são armazenados no repositório da LP. Durante a fase de Engenharia de Aplicação, são fornecidos os requisitos da aplicação e utilizado os modelos armazenados no repositório para instanciar um produto da LP. Caso ocorra algum problema de requisitos e/ou erros, será retomada a atividade de Engenharia da LP em que serão corrigidos esses problemas, caso contrário, é gerado uma aplicação executável iniciando a fase de teste da aplicação. Na fase de teste FSM-TSPL, o Testador (Projetista de teste) cria as especificações das MEFs representado os produtos a serem testados e informa o conjunto de testes definida pelo usuário. As MEFs não foram criadas na Engenharia de Domínio, pelo fato de, caso as mesmas contenham erros, estes podem ser repassados para todos os outros produtos. Após a aplicação dos testes é entregue a aplicação ao Cliente.

A base para a estratégia de teste incremental é a utilização de algoritmos para a geração das seqüências de testes, as quais são baseadas nas especificações das MEFs. Sendo assim, nossa abordagem utiliza dois algoritmos o quais usam o método HSI (Luo et al., 1994) e método P (Simão e Petrenko, 2010), para a geração dos casos de teste.

Considere a AGM sob teste, de forma que, um conjunto de produtos pode ser instanciado, nos referimos ao produto da LP como $prod_i$. Na FSM-TSPL, assumimos que, depois da instanciação, um produto é testado usando uma MEF. A notação M_i representa a MEF para o produto $prod_i$. Assumimos que há um conjunto de teste inicial TS_i derivado de M_i. Este pode ser gerado usando qualquer método ou uma abordagem *ad hoc*. O processo de teste sempre ocorre em pares, ou seja, os casos de testes são gerados a partir de uma especificação M_i e utilizados para testar uma implementação M_j, desta forma, podemos reutilizar os testes gerados anteriormente e incrementá-los para atender a implementação.

O processo da FSM-TSPL é dividido nos seguintes passos, conforme representado pela Figura 4.4:

1. *Criar MEF para cada produto*: deve ser criada uma MEF M_i para cada produto $prod_i$ instanciada a partir da LP. As MEFs podem ser criadas pelo testador ou derivadas a partir dos artefatos da LP. As MEFs desenvolvidas para os produtos da LP são, usualmente, determinísticas e parciais, dado grande o número de entradas e transições, devido as variabilidades da LP.

2. *Selecionar uma MEF*: após a criação da especificação dos produtos, o testador deve

selecionar uma MEF para realizar o teste.

3. *Informar conjunto de teste definido pelo usuário*: este passo ocorre apenas se o testador quiser utilizar um conjunto de teste (casos de testes de um outro produto), caso contrário serão gerados casos de testes a partir de uma seqüência vazia ϵ.

4. *Gerar conjunto de teste com o método HSI*: é utilizado o método HSI para a geração de casos de teste para cada produto da LP. Este passo somente é realizado caso não seja atribuído um conjunto de teste pelo testador.

5. *Verificar o conjunto de teste*: após a geração do conjunto de teste TS_i, deve ser verificado se os casos de teste gerados pela MEF M_i (especificação) também são definidos na MEF M_j (implementação).

6. *Refinar o conjunto de teste*: Caso haja algum caso de teste não definido na MEF M_j o mesmo deve ser removido, ou seja, $TS_j = TS_i \cap \Omega(M_j)$.

7. *Gerar conjunto de teste com o método P*: após a verificação dos casos de teste, estes serão utilizados para a geração de novos casos de teste utilizando o método P. Desta a forma, o método P reutiliza os casos de teste gerados anteriormente e garante, na maioria das vezes, a geração de um novo conjunto de testes menores.

8. *Comparar ambos conjuntos de teste*: deve realizar a comparação dos casos de teste gerados pelo método HSI e método P. Desta forma, é possível avaliar a eficiência do método P e a viabilidade da utilização do mesmo. Caso queira testar uma nova MEF volte para o passo 2, quando um novo produto $prod_k$ será instanciado da LP e testado.

De acordo com o passo 1, foram criadas as MEFs M_1 e M_2 especificando os produtos $prod_1$ e $prod_2$ respectivamente, ambos instanciados a partir da AGM. A MEF M_1 contém apenas características comuns e representa a especificação do jogo Brickles, esta ilustrada pela Figura 4.5. A MEF também pode ser representada pela tupla $M = (\{SG, SB, PS\}, SG, \{SG, PS, SV, EX\}, \{0, 1\}, D_M, \delta, \lambda)$, contendo 3 estados, 4 entradas, 2 saídas, 12 transições.

De maneira a facilitar a especificação de MEFs, foram utilizadas duas DSLs, o formato KISS ([1]) e o formato RFSM (*Ruby Finite State Machine*). A DSL RFSM foi implementada neste trabalho, utilizando a linguagem Ruby e sua definição será apresentado na Tabela 4.1.

A especificação da MEF utilizando o formato KISS ocorre através da definição de transições. Cada transição é definida em uma nova linha, especificando o estado atual, entrada, saída e próximo estado, estes são separados por alguns caracteres especiais. Dado a definição da primeira linha da Listagem 4.1, o estado inicial SG é seguido pelo caractere $--$ que separa

[1] É um formato tabular, cada linha tem quatro entradas: o estado atual, a entrada, a saída e o próximo estado.

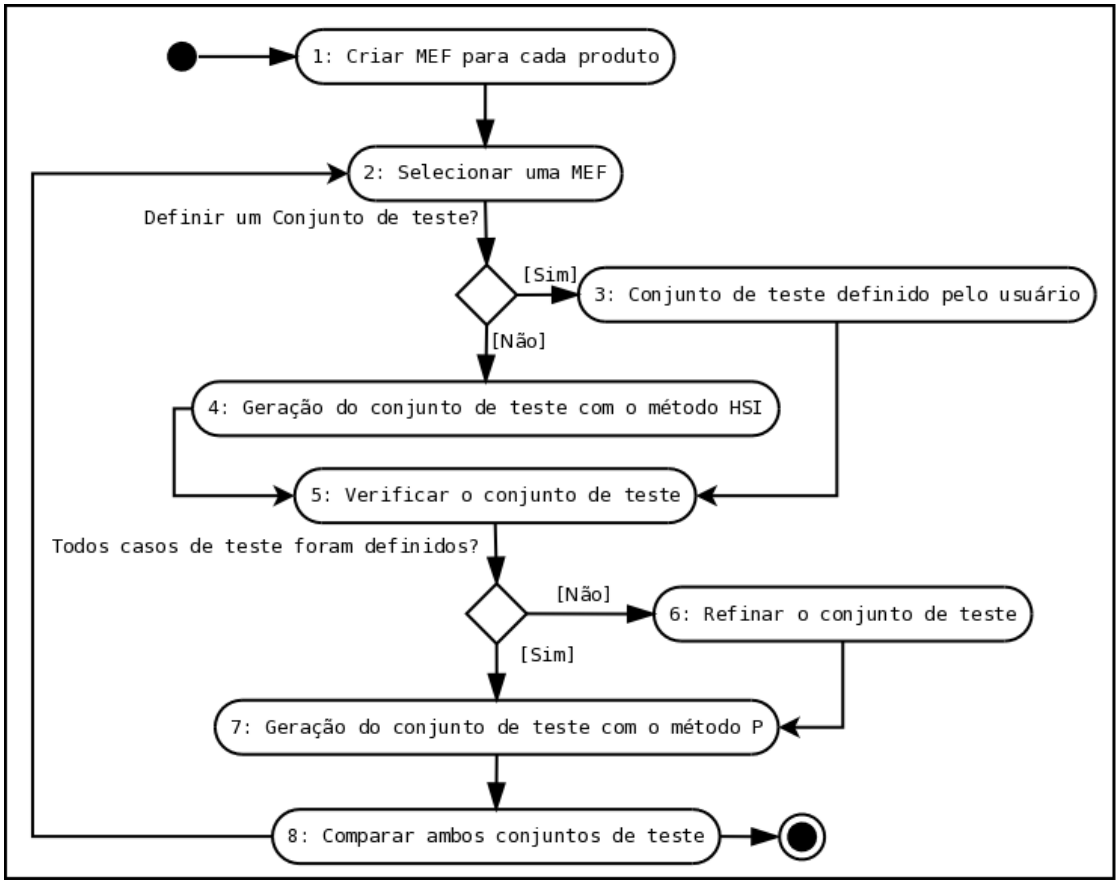

Figura 4.4: O processo da estratégia FSM-TSPL.

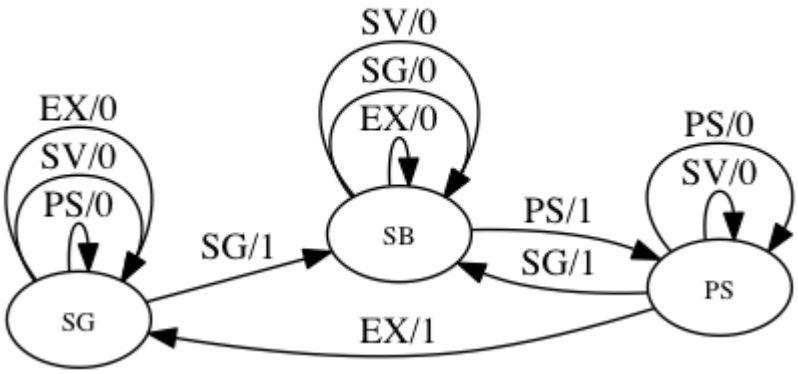

Figura 4.5: MEF do produto 1 da LP AGM

este da entrada EX, em seguida o caractere / separa este da saída 0, por último o caractere $->$ separa a saída do próximo estado SG.

```
SG -- EX / 0 -> SG
SG -- SV / 0 -> SG
SG -- PS / 0 -> SG
SG -- SG / 1 -> SB
SB -- SG / 0 -> SB
SB -- PS / 1 -> PS
SB -- EX / 0 -> SB
SB -- SV / 0 -> SB
PS -- PS / 0 -> PS
PS -- SG / 1 -> SB
PS -- EX / 1 -> SG
PS -- SV / 0 -> PS
```

Listagem 4.1: Especificação da MEF M_1 no formato KISS

A Listagem 4.2 ilustra a especificação da MEF com o formato RSFM, cuja especificação deve estar de acordo com os elementos definidos pela Tabela 4.1.

```
mef "ExpBk1" do
  state_initial "SG"
  domain "SG -- EX / 0 -> SG"
  domain "SG -- SV / 0 -> SG"
  domain "SG -- PS / 0 -> SG"
  domain "SG -- SG / 1 -> SB"
  domain "SB -- SG / 0 -> SB"
  domain "SB -- PS / 1 -> PS"
  domain "SB -- EX / 0 -> SB"
  domain "SB -- SV / 0 -> SB"
  domain "PS -- PS / 0 -> PS"
  domain "PS -- SG / 1 -> SB"
  domain "PS -- EX / 1 -> SG"
  domain "PS -- SV / 0 -> PS"

  format_test_case
  generate_sequences
  generate_mef_graph
  generate_tree_graph
  generate_divergence_graph
end
```

Listagem 4.2: Especificação da MEF M_1 no formato RFSM

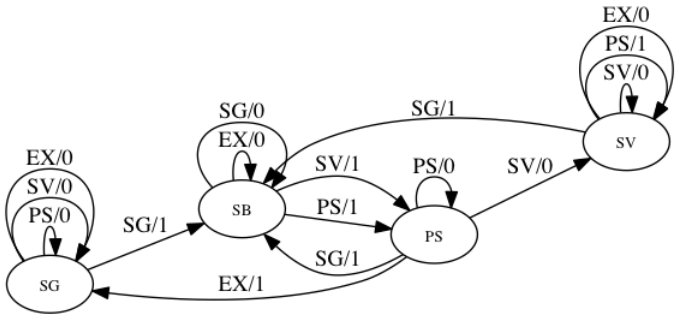

Figura 4.6: MEF do produto 2 da LP AGM

A MEF M_2, também especifica o jogo Brickles e contém as características comuns e uma característica opcional, *save game*, representada pela Figura 4.6. Esta também pode ser

Tabela 4.1: Tabela de Definição da DSL RFSM

Primitiva	Parâmetros	Obrigatório	Descrição
mef	name, block	Sim	Elemento principal da especificação. Define a MEF, é responsável pela definição dos outros elementos. O parâmetro "name" nomeia a MEF, é utilizado na geração dos grafos. O "block" é uma referência para todos os elementos especificados dentro do bloco "do .. end" da MEF.
state_initial	state	Sim	Elemento que define qual será o estado inicial da MEF.
domain	domain	Sim	Define um domínio da MEF (estado inicial, entrada, saída e estado final).
no_format_test_case			Define o formato da seqüência de testes, sem separação (Default).
format_test_case			Define o formato da seqüência de testes, sendo os mesmos separados por vírgula.
generate_sequences		Sim	Responsável por gerar a seqüência de testes HSI, baseado nos domínios e estado inicial, definidos na especificação.
generate_mef_graph			Responsável por gerar o grafo da MEF, este gerado nos formatos "DOT" e "PNG".
generate_tree_graph			Responsável por gerar o grafo da árvore de teste, este gerado nos formatos "DOT" e "PNG".
generate_divergence_graph			Responsável por gerar o grafo de distinção, este gerado nos formatos "DOT", "SVG".

representada pela tupla $M = (\{SG, SB, PS, SV\}, SG, \{SG, PS, SV, EX\}, \{0, 1\}, D_M, \delta, \lambda)$, contendo 4 estados, 4 entradas, 2 saídas, 16 transições. A especificação da MEF no formato KISS e RFSM é ilustrado pelas Listagens 4.3 e 4.4, respectivamente.

Após a criação das MEFs, selecionamos a MEF M_1 para a realização dos testes, conforme passo 2. Como a MEF M_1 é o primeiro produto a ser testado, pulamos o passo 3, não definindo um conjunto de teste, desta forma o mesmo será gerado no próximo passo. A MEF M_1 é

utilizada como entrada para o método HSI, o qual é responsável por gerar o conjunto de teste, conforme descrita no passo 4.

Para realizar a geração dos conjuntos de testes do método HSI, foi implementado o algoritmo na linguagem Ruby ([2]), permitindo a especificação de MEFs nas duas DSLs, formato KISS e RFSM. O programa permite a geração de conjunto de testes HSI para MEFs parciais e mínimas.

```
SG -- EX / 0 -> SG
SG -- SV / 0 -> SG
SG -- PS / 0 -> SG
SG -- SG / 1 -> SB
SB -- SG / 0 -> SB
SB -- PS / 1 -> PS
SB -- EX / 0 -> SB
SB -- SV / 1 -> PS
PS -- PS / 0 -> PS
PS -- SG / 1 -> SB
PS -- EX / 1 -> SG
PS -- SV / 0 -> SV
SV -- EX / 0 -> SV
SV -- PS / 1 -> SV
SV -- SG / 1 -> SB
SV -- SV / 0 -> SV
```

Listagem 4.3: Especificação da MEF M_2 no formato KISS

```
mef "ExpBk2" do
  state_initial "SG"
  domain "SG -- EX / 0 -> SG"
  domain "SG -- SV / 0 -> SG"
  domain "SG -- PS / 0 -> SG"
  domain "SG -- SG / 1 -> SB"
  domain "SB -- SG / 0 -> SB"
  domain "SB -- PS / 1 -> PS"
  domain "SB -- EX / 0 -> SB"
  domain "SB -- SV / 1 -> PS"
  domain "PS -- PS / 0 -> PS"
  domain "PS -- SG / 1 -> SB"
  domain "PS -- EX / 1 -> SG"
  domain "PS -- SV / 0 -> SV"
  domain "SV -- EX / 0 -> SV"
  domain "SV -- PS / 1 -> SV"
  domain "SV -- SG / 1 -> SB"
  domain "SV -- SV / 0 -> SV"

  format_test_case
  generate_sequences
  generate_mef_graph
  generate_tree_graph
  generate_divergence_graph
end
```

Listagem 4.4: Especificação da MEF M_2 no formato RFSM

4.2.1 Geração da seqüência de testes com o método HSI

O processo de execução do algoritmo HSI, usando uma MEF no formato KISS, ocorre da seguinte forma, representada pela Figura 4.7:

[2]Código disponível em: http://github.com/maykon/MetodoHSI

1. *Especificação da MEF*: Deve ser gerado um arquivo contendo a especificação da MEF no formato KISS.

2. *Análise sintática da MEF*: nesta etapa é analisada a estrutura sintática do arquivo, garantindo que o conteúdo esteja no formato correto, caso contrário, o programa é encerrado.

3. *Tradução da MEF*: O algoritmo percorre linha a linha e gera a tradução da MEF, atribuindo o conjunto de estados, entradas, saídas e transições.

4. *Geração da Sequência de Testes HSI*: Concluída a tradução, são criados os conjuntos necessários para a geração do conjunto de teste HSI (conjunto H, cobertura de estados e cobertura de transição), e a seqüência de teste em si.

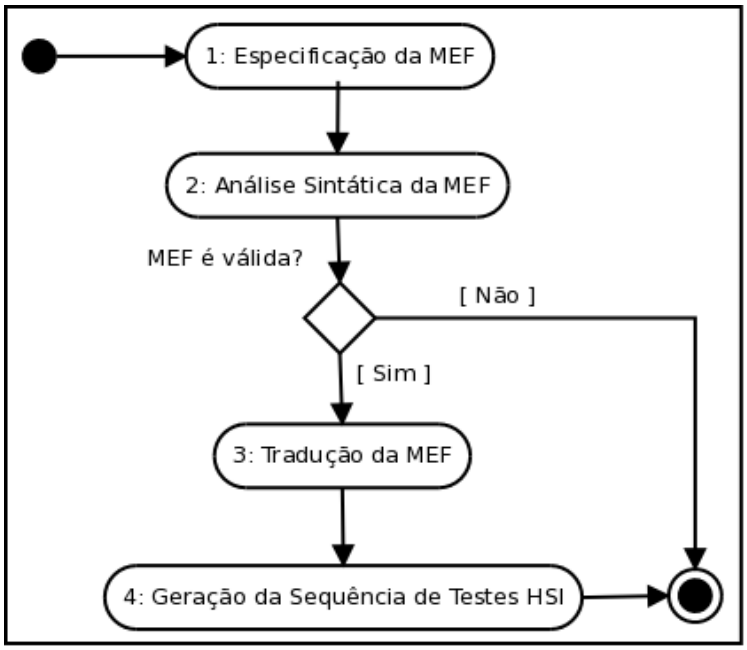

Figura 4.7: Diagrama de Atividades do Algoritmo HSI - KISS

A execução do algoritmo HSI que utiliza o formato RFSM, funciona de forma semelhante ao anterior, a principal diferença é que o código da especificação é um código executável, ao contrário do formato KISS que é interpretado. Além disto, por meio desta especificação, o testador pode gerar os grafos da MEF, grafo da árvore de teste da MEF e o gráfico de distinção.

A execução deste algoritmo é apresentada nos passos abaixo, ilustrada pela Figura 4.8:

1. *Especificação da MEF*: Deve ser gerado a especificação da MEF no formato da DSL em Ruby, o qual deve conter todos os elementos obrigatórios para a correta tradução da MEF.

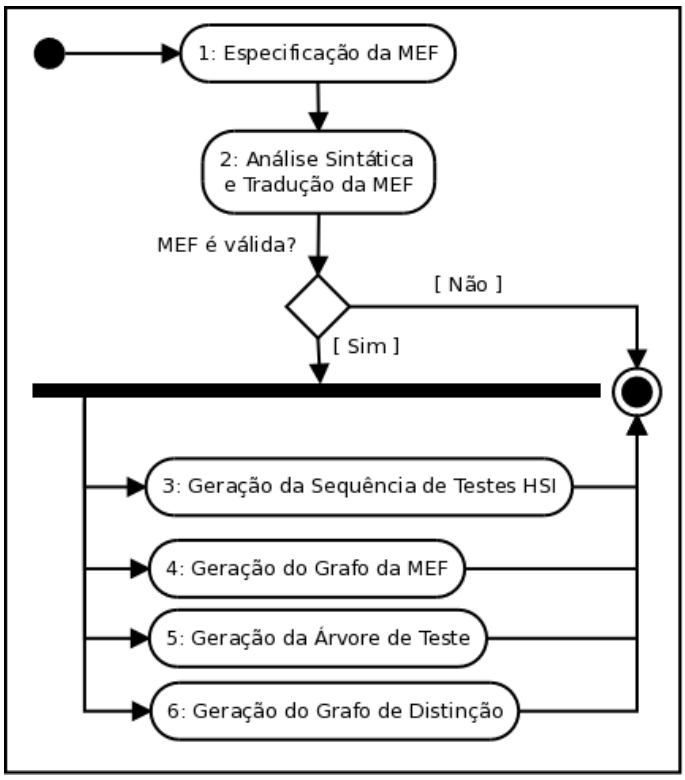

Figura 4.8: Diagrama de Atividades do Algoritmo HSI - Ruby

2. *Análise sintática e Tradução da MEF*: A análise sintática é realizada em duas etapas, durante a execução da especificação, como o código é executável, e pelo interpretador Ruby. Na segunda etapa ocorre a análise sintática e tradução da MEF, a qual é realizada na execução das primitivas *mef*, *state_ initial* e *domain* da DSL.

3. *Geração da Sequência de Testes HSI*: a geração da seqüência de testes ocorre exatamente como no algoritmo anterior, são gerados os conjuntos de testes necessários para a criação da seqüência de testes HSI, os quais são baseados na especificação da MEF, definidas pelos elementos *state_ initial* e *domain*. Os conjuntos de testes são impressos na tela, os quais podem ser representados em dois formatos diferentes, sem separação ou separados por vírgula.

4. *Geração do Grafo da MEF*: A geração do grafo da MEF ocorre quando a primitiva *generate_ mef_ graph* é utilizada. Esta gera a especificação do grafo na linguagem DOT, o arquivo da especificação e uma imagem no formato PNG.

5. *Geração da árvore de Teste*: A árvore de testes é gerada na execução do elemento *generate_ tree_ graph*, igualmente a geração do grafo da MEF é gerada a especificação da árvore de testes no formato DOT e uma imagem em PNG.

6. *Geração do Grafo de Distinção*: O grafo de distinção é gerada na chamada da primitiva *generate_ divergence_ graph*, neste caso, é gerado a especificação em DOT e um arquivo

no formato SVG. Esta escolha ocorre devido a possibilidade de ser gerado uma figura muito grande, desta forma, este possui uma melhor resolução e o tamanho do arquivo é menor comparado a outros formatos.

Após a definição da MEF M_1 foi aplicado o método HSI, o qual gerou os seguintes conjuntos: $Q = \{\ \epsilon;\ SG;\ "SG, PS"\ \}$, $P = \{\ \epsilon;\ SG;\ "SG, PS",\ PS,\ SV,\ EX,$ $"SG, SG",\ "SG, SV",\ "SG, EX",\ "SG, PS, SG",\ "SG, PS, PS",\ "SG, PS, SV",$ $"SG, PS, EX"\ \}$ e os conjuntos H_i são: $H_0 = \{\ EX,\ PS\ \}$, $H_1 = \{\ EX,\ PS\ \}$ e $H_2 = \{\ EX\ \}$.

De posse destes conjuntos é gerado os casos de teste TS_{HSI} representado pela Listagem 4.5, o qual possui tamanho *72*.

```
EX,EX
EX,PS
PS,EX
PS,PS
SG,EX,EX
SG,EX,PS
SG,PS,EX,EX
SG,PS,EX,PS
SG,PS,PS,EX
SG,PS,SG,EX
SG,PS,SG,PS
SG,PS,SV,EX
SG,SG,EX
SG,SG,PS
SG,SV,EX
SG,SV,PS
SV,EX
SV,PS
```

Listagem 4.5: Conjunto de testes MEF M_1 com formatação

```
EX,EX
EX,PS
EX,SV
PS,EX
PS,PS
PS,SV
SG,EX,EX
SG,EX,SV
SG,PS,EX
SG,SG,EX
SG,SG,SV
SG,SV,EX,EX
SG,SV,EX,PS
SG,SV,EX,SV
SG,SV,PS,EX
SG,SV,SG,EX
SG,SV,SG,SV
SG,SV,SV,EX,EX
SG,SV,SV,EX,PS
SG,SV,SV,EX,SV
SG,SV,SV,PS,EX
SG,SV,SV,PS,PS
SG,SV,SV,PS,SV
SG,SV,SV,SG,EX
SG,SV,SV,SG,SV
SG,SV,SV,SV,EX
SG,SV,SV,SV,PS
SG,SV,SV,SV,SV
SV,EX
SV,PS
SV,SV
```

Listagem 4.6: Conjunto de testes MEF M_2 usando o método HSI

Aplicando o método HSI na MEF M_2 obtemos os seguintes conjuntos de teste: $Q = \{\ \epsilon;\ SG;\ "SG, SV"\ "SG, SV, SV"\ \}$, $P = \{\ \epsilon;\ SG;\ "SG, SV",\ "SG, SV, SV",\ SV,\ PS,\ EX,\ "SG, PS",\ "SG, SG",\ "SG, EX",\ "SG, SV, SG",\ "SG, SV, PS",\ "SG, SV, EX",\ "SG, SV, SV, PS",\ "SG, SV, SV, SV",\ "SG, SV, SV, SG",\ "SG, SV, SV, EX"\}$ e os conjuntos H_i são: $H_0 = \{\ EX,\ PS\ SV\}$, $H_1 = \{\ EX,\ SV\ \}$, $H_2 = \{\ EX,\ PS,\ SV,\ \}$ e $H_3 = \{\ EX\ \}$.

Os casos de testes gerados para a MEF M_2 é representado pela Listagem 4.6, de tamanho 143.

A geração dos grafos, é apoiada apenas pelo formato RFSM, a representação da árvore de teste da MEF M_1 é ilustrado pela Figura 4.9 e o Grafo de Distinção é representado pela Figura 4.10.

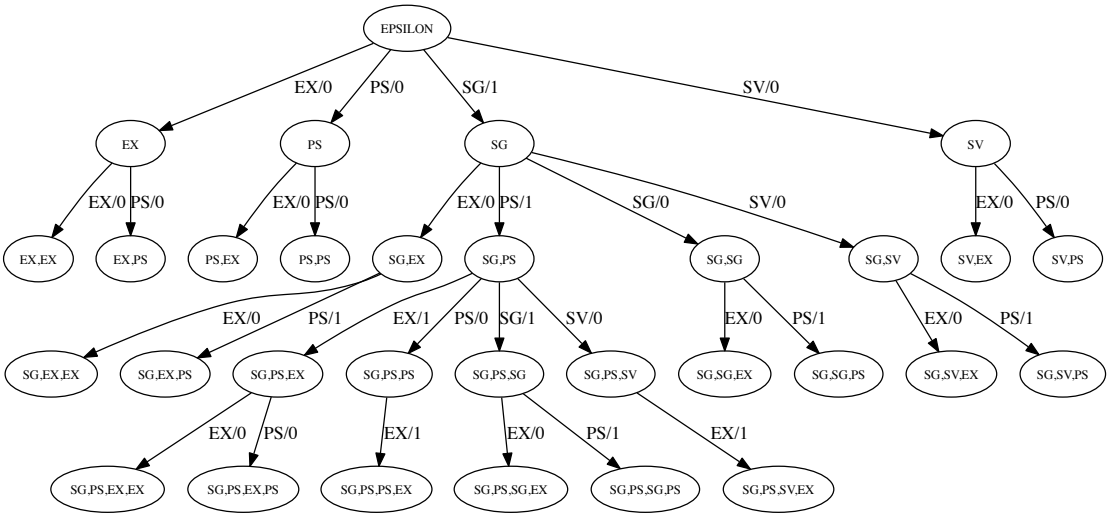

Figura 4.9: Árvore de teste da MEF M_1

Após a geração das seqüências de testes da MEF M_1, selecionamos a MEF M_2, conforme passo 2, e utilizamos as seqüências de testes gerados a partir da MEF M_1, desta forma, verificamos se os conjuntos de testes são especificados na MEF M_2, conforme passo 5. No passo 6 são removido as seqüências de testes que não são especificados pela MEF M_2. Isto ocorre quando algumas seqüências de testes não estão definidas na MEF de implementação, neste caso M_2, desta forma é necessário retirá-las, diminuindo o custo de testar a implementação.

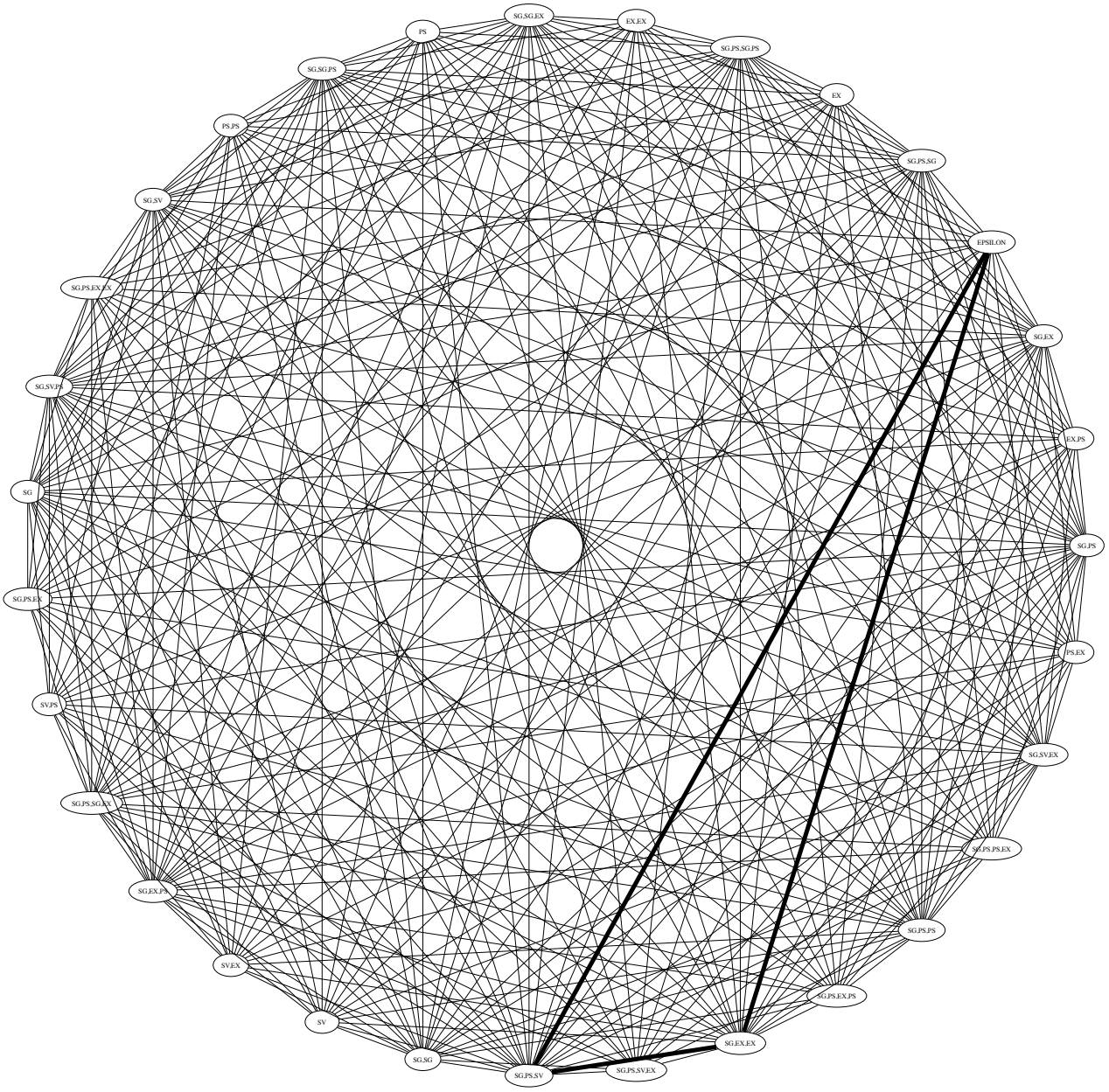

Figura 4.10: Grafo de distinção da MEF M_1

4.2.2 Geração da seqüência de testes com o método P

A geração dos conjuntos de testes do método P, passo 7, utiliza um algoritmo implementado na linguagem Java ([3]), o qual possui como entrada um arquivo contendo a especificação da MEF no formato KISS. Caso seja especificado um conjunto de testes inicial, o método P incrementa o conjunto de teste até que o mesmo consiga cobrir a MEF, caso contrário, este gerará o conjunto de testes a partir do ϵ.

O funcionamento do algoritmo do método P é descrito abaixo, conforme representado pela Figura 4.11:

1. *Especificação da MEF*: A especificação da MEF ocorre da mesma maneira que a do algoritmo HSI usando o formato KISS. O algoritmo permite mais dois parâmetros opcionais, o primeiro define o conjunto de teste que será incrementado e o segundo refere-se ao formato de saída da seqüência de teste.

2. *Análise Sintática da MEF*: A análise sintática ocorre da mesma maneira que a do algoritmo HSI.

3. *Tradução da MEF*: A tradução da MEF também ocorre da mesma forma que a do algoritmo HSI.

4. *Geração da Sequência de Testes incrementando o conjunto*: Nesta etapa, o usuário define o conjunto de teste o qual será incrementado até que sejam gerados casos de testes capazes de cobrirem a MEF.

5. *Geração da Sequência de Testes a partir do ϵ*: Caso o usuário não informe os casos de teste, os mesmos serão gerados a partir da seqüência vazia (ϵ), até que satisfaça a cobertura da MEF.

Para a geração dos conjuntos de testes da MEF M_2, passo 7, utilizamos o testes gerados da MEF M_1, de forma que o método P utiliza estes testes e os incrementa até atingir o nível de cobertura satisfatórios, as seqüências de testes gerados pelo método P para a MEF M_2 é representado pela Listagem 4.7.

Após a geração dos casos de teste usando o método HSI, aplicamos o método P na MEF M_2 e utilizamos os casos de testes da MEF M_1, gerados pelo HSI, como entrada para o conjunto de teste do método P. Desta forma obtemos os seguintes casos de teste representados pela Listagem 4.7, estes de tamanho *104*.

[3]Criado por Andre Takeshi Endo, doutorando em Ciência da Computação pelo ICMC-USP/São Carlos

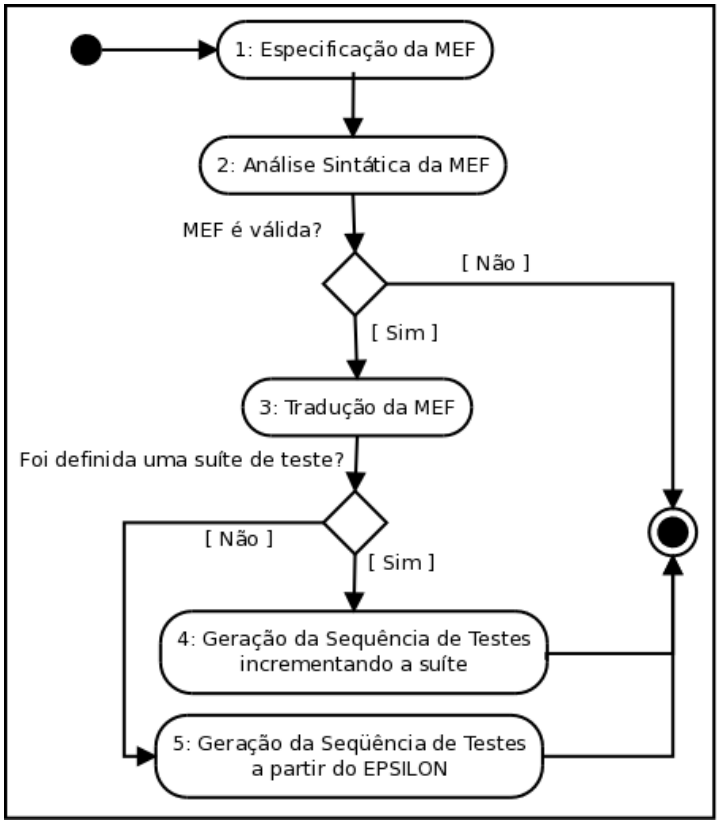

Figura 4.11: Diagrama de Atividades do algoritmo P

```
EX,EX
EX,PS
PS,EX
PS,PS
SG,EX,EX
SG,PS,EX,EX
SG,PS,EX,PS
SG,PS,PS,EX
SG,PS,SG,EX
SG,PS,SV,EX
SG,SG,EX
SG,SV,EX
SG,SV,PS
SV,EX
SV,PS
SG,SG,PS,EX
SG,EX,PS,EX
SG,PS,SG,PS,EX
SG,SV,SV,EX,PS,EX
SG,SV,SV,PS,EX,PS,EX
SG,SV,SV,SG,PS,EX
SG,SV,SV,SV,PS,EX
```

Listagem 4.7: Conjunto de testes MEF M_2 com formatação

O último passo é a comparação dos casos de testes gerados pelo método HSI e P, desta forma, analisando o custo da geração dos casos de testes, podemos observar que os casos de teste gerados pelo método HSI possui tamanho *143*, enquanto os casos de testes gerados pelo método P possui tamanho *104* sendo estes menores que o método HSI.

4.3 Considerações Finais

Neste capítulo, apresentamos a estratégia de geração de teste incremental baseado em MEFs, o qual consiste na utilização de algoritmos que são responsáveis pela geração do conjunto de testes. Esta estratégia permite a geração de testes incrementais podendo começar a partir de uma seqüência vazia ou através de um conjunto de testes definida pelo usuário.

No próximo Capítulo, será apresentado avaliação da estratégia de teste FSM-TSLP, o qual foi utilizado dois experimentos de maneira a analisar a viabilidade da estratégia proposta.

Capítulo

5

Avaliação da Estratégia

5.1 Considerações Iniciais

Neste capítulo são apresentados os resultados obtidos por meio de dois experimentos, AGM e Mobile Media. De maneira à avaliar a viabilidade da estratégia proposta, os experimentos foram conduzidos de forma a avaliar o tamanho do conjunto de testes (custo de geração do conjunto de testes) e o número de operações resets. Foram instanciados alguns produtos da LP para cada experimento, sendo estes, especificados por meio de MEFs. As MEFs foram modeladas manualmente baseada no comportamento de cada produto instanciado da LP, sendo que estas possuem as seguintes propriedades: completas, determinísticas, mínimas e inicialmente conexas.

Os algoritmos utilizados na avaliação dos experimentos foram executados em um computador Intel(R) Core(TM)2 Duo, 2.4GHz, 4GB de RAM e sistema operacional Mac OS X 10.6.8.

O primeiro experimento, descrito na Seção 5.2, apresenta os resultados do experimento da LP AGM, utilizada como ilustração no Capítulo anterior, que aborda uma LP contendo alguns jogos para dispositivos móveis. O segundo experimento, descrito na Seção 5.3, apresenta os resultados do experimento *Mobile Media*(Figueiredo et al., 2008), sendo esta uma LP com diversas funcionalidades de multimídia para dispositivos móveis.

A metodologia utilizada nos estudos experimentais pode ser descrita pelas seguintes etapas:

1. Criação das MEFs para cada produto da LP

2. Selecionar uma MEF para ser testada
3. Aplicar o método HSI na MEF selecionada
4. Aplicar o método P na MEF selecionada
5. Aplicar a estratégia FSM-TSPL utilizando o conjunto de testes gerados pelo HSI na MEF selecionada
6. Aplicar a estratégia FSM-TSPL utilizando o conjunto de testes gerados pelo P na MEF selecionada
7. Refinar conjunto de teste gerado pela FSM-TSPL nos conjuntos HSI
8. Refinar conjunto de teste gerado pela FSM-TSPL nos conjuntos P
9. Comparação dos conjuntos gerados
10. Análise dos resultados.

5.2 Experimento 1: AGM

O experimento usando a LP AGM foi conduzido com o objetivo de analisar o comportamento da estratégia de teste proposta FSM-TSPL. A partir da AGM foram instanciados 6 produtos contendo características distintas, estes foram instanciados a partir da Base (características obrigatórias) e para cada produto foram selecionadas características alternativas e opcionais, conforme podemos analisar na Tabela 5.1.

Tabela 5.1: Produtos instanciados da AGM

Produto	Características
Prod.1	Base + BK
Prod.2	Base + BK + SV
Prod.3	Base + PG
Prod.4	Base + PG + SV
Prod.5	Base + BW
Prod.6	Base + BW + SV

Legenda		
	BK	Brickles
	PG	Pong
	BW	Bowling
	SV	Save

Para cada produto foi gerada uma MEF, o número de estados das MEFs variam entre 3 e 4, estas possuem 4 entradas e 2 saídas cada. Todas as MEFs são mínimas, completas,

determinísticas e inicialmente conexas, conforme apresentado na Tabela 5.2.

Tabela 5.2: MEFs representando os produtos AGM

MEF	Estados	Entradas	Saídas
Prod.1	3	4	2
Prod.2	4	4	2
Prod.3	3	4	2
Prod.4	4	4	2
Prod.5	3	4	2
Prod.6	4	4	2

Na Tabela 5.3 são apresentados os custos de geração dos conjuntos de teste gerados a partir das MEFs utilizando os métodos HSI, método P (utilizando um conjunto de teste vazio ϵ) e a estratégia FSM-TSPL. Na estratégia FSM-TSPL, foram iniciados os testes com o conjunto HSI e P respectivamente, os conjuntos de testes foram gerados da seguinte forma: para o produto 1 foram considerados os conjuntos gerados pelos métodos HSI e P, desta forma foi desconsiderado seu custo; o produto 2 foi testado com os conjuntos gerados a partir do produto 1; o produto 3 a partir do produto 2 e assim respectivamente. Os conjuntos de testes gerados pela estratégia FSM-TSPL foram refinados (removidas duplicidades, seqüências definidas no conjunto de entrada, e prefixos), desta forma gerando um conjunto de testes relativamente menor.

Tabela 5.3: Custo de geração dos conjuntos

Produto/Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	72	48	–	–
Prod.2	143	80	45	45
Prod.3	64	64	48	42
Prod.4	120	96	55	54
Prod.5	60	48	23	28
Prod.6	108	66	52	64

Podemos notar que o método HSI gerou os maiores conjuntos comparado com os outros métodos. Enquanto que, o método P apresentou conjuntos relativamente menores que o HSI. Também podemos analisar que os conjuntos de teste gerados a partir da estratégia FSM-TSPL foram menores se comparados com os outros métodos. Também podemos analisar que em alguns casos, os conjuntos gerados usando a estratégia FSM-TSPL(P) são maiores que os gerados pela FSM-TSPL(HSI), conforme visto nos produtos 5 e 6. Isto pode ocorrer em alguns casos, quando o método P não consegue reduzir os casos de testes informados. A Figura 5.1 ilustra a relação entre o custo dos conjuntos de teste gerados para cada produto/método,

conforme visto anteriormente.

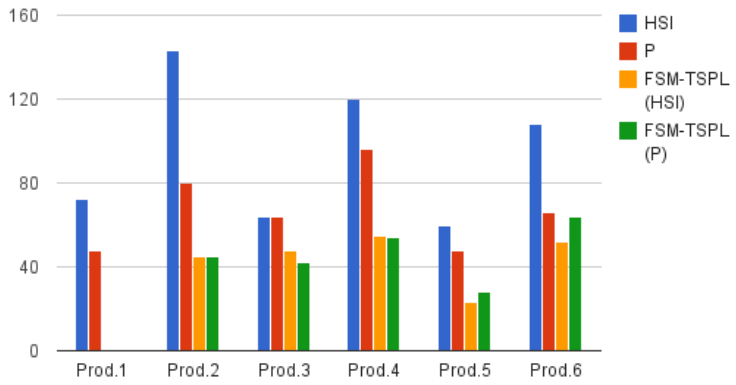

Figura 5.1: Custo da Geração das Seqüências x Produtos/Métodos

A quantidade de operações reset também foi calculada com base nos conjuntos de testes refinados e estes são apresentados na Tabela 5.4. A relação entre a quantidade de resets para cada produto/método pode ser analisado pela Figura 5.2. O método HSI também gerou mais operações reset comparado com os outros métodos, enquanto a FSM-TSPL gerou o menor número de operações resets. Conforme analisado anteriormente, no custo de geração dos conjuntos, em alguns casos a estratégia FSM-TSPL(P), não consegue reduzir algumas seqüências de testes, desta forma, o número de resets é maior se comparada com o número de resets da FSM-TSPL(HSI).

Tabela 5.4: Quantidade de operações reset

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	18	11	–	–
Prod.2	31	15	7	7
Prod.3	15	15	7	8
Prod.4	26	19	10	10
Prod.5	16	13	5	6
Prod.6	26	13	10	12

Na Tabela 5.5 são apresentados o comprimento das sequências por produtos/métodos, este também é ilustrado pela Figura 5.3. O comprimento das seqüências geradas pelo FSM-TSPL, é maior ou igual comparada aos outros métodos, pois este gera seqüências maiores de maneira a diminuir a quantidade de operações resets.

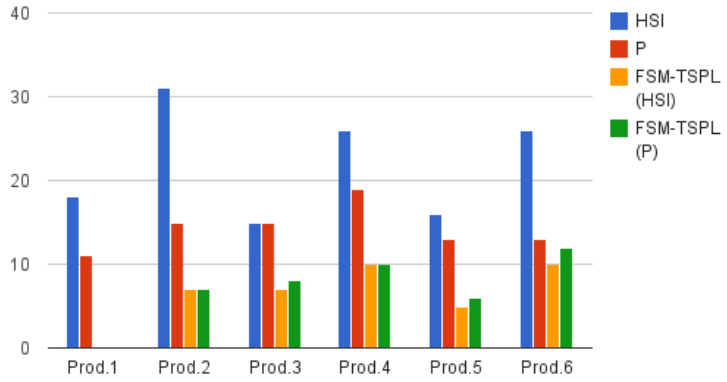

Figura 5.2: Número Resets x Produtos/Métodos

Tabela 5.5: Comprimento das seqüências

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	3	3	–	–
Prod.2	3	4	5	5
Prod.3	3	3	5	4
Prod.4	3	4	4	4
Prod.5	2	2	3	3
Prod.6	3	4	4	4

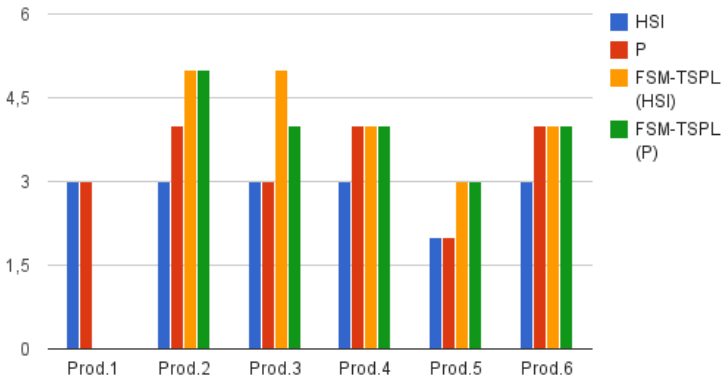

Figura 5.3: Comprimento das Seqüências x Produtos/Métodos

A Tabela 5.6 apresenta os dados referentes ao tempo médio de execução dos algoritmos em milisegundos. O método HSI possui os menores tempos de geração de seqüências, enquanto que a estratégia FSM-TSPL possui os maiores tempos, visto que estes analisam os conjuntos de testes utilizados como entrada e os incrementa conforme for necessário, desta forma gastando maior tempo para sua completa execução, conforme podemos observar a Figura 5.4.

Tabela 5.6: Tempo médio de execução dos algoritmos em *ms*

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	15	553	–	–
Prod.2	71	1504	1941	1671
Prod.3	13	552	3495	2966
Prod.4	64	1531	1598	1485
Prod.5	6	448	2205	2695
Prod.6	18	1261	1733	1758

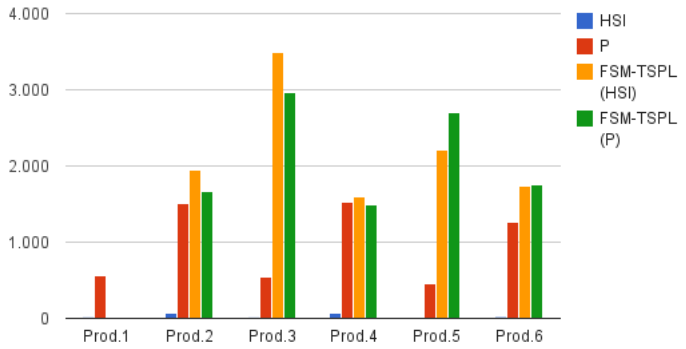

Figura 5.4: Tempo médio de execução dos algoritmos em *ms*

A Tabela 5.7 apresenta os custos acumulativos referente a geração dos conjuntos de testes por produto/método, este também é ilustrada pela Figura 5.5. Os custos foram calculados pela soma dos custos de todos os produtos relacionados com cada método. Conforme podemos observar, o método HSI possui o maior custo para testar todos os produtos, seguido do P e FSM-TSPL (P). O FSM-TSPL (HSI) apresenta o menor custo sendo este de apenas *223*.

A utilização da estratégia FSM-TSPL (HSI) obteve uma economia de *61%* comparado com o HSI e *45%* comparado com o método P. Já a FSM-TSPL (P) obteve *59%* comparado com o HSI e *42%* comparado com o P.

Tabela 5.7: Custo acumulativo dos conjuntos de teste

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	72	48	–	–
Prod.1,2	215	128	45	45
Prod.1,2,3	279	192	93	87
Prod.1,2,3,4	399	288	148	141
Prod.1,2,3,4,5	459	336	171	169
Prod.1,2,3,4,5,6	567	402	223	233

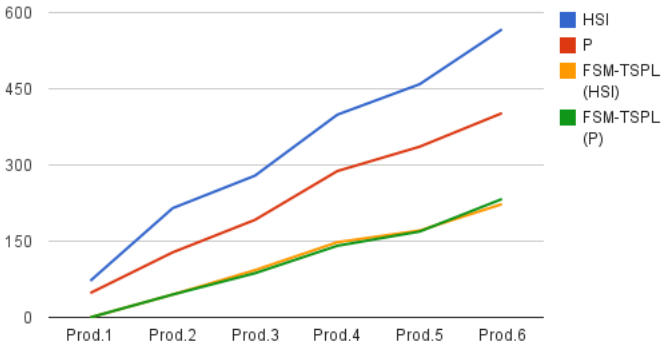

Figura 5.5: Custo Acumulativo da Geração das Seqüências x Produtos/Métodos

5.3 Experimento 2: Mobile Media

Mobile Media (Figueiredo et al., 2008) é uma LP contendo diversas características como: manipulação de fotos, músicas e vídeos em dispositivos móveis, desenvolvida com base na LP MobilePhoto (Young e Young, 2005). Esta LP foi estendida para conter novas características obrigatórias, alternativas e opcionais, conforme a Figura 5.6.

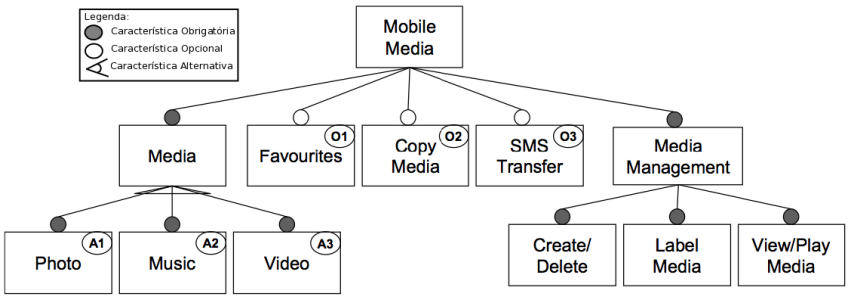

Figura 5.6: Modelo de Características Mobile Media (Figueiredo et al., 2008)

Conforme podemos observar, a LP Mobile Media possui características obrigatórias (criar/apagar mídia, rotular mídia, e visualizar/tocar mídia), características alternativas (tipos de mídia suportadas: foto, música e/ou vídeos) e algumas características opcionais que são: transferir fotos por SMS, contar e ordenar mídia, copiar mídia e conjunto de favoritos.

A Mobile Media incorporou sete alterações em cenários, o que levou a oito entregas (releases). Este cenário compreende várias mudanças envolvendo características obrigatórias, alternativas, opcionais e interesses não funcionais conforme representado pela Tabela 5.8 (Figueiredo et al., 2008).

A partir da Mobile Media foram instanciados 24 produtos contendo características distintas, de forma que os produtos foram instanciados a partir da Base (características obrigatórias) e para cada produto foram selecionadas características alternativas e opcionais, conforme podemos analisar na Figura 5.7.

Para cada produto, foi gerado uma MEF, de maneira que o número de estados das MEFs variam entre 3 a 6, estas possuem 8 entradas e 2 saídas cada. Todas as MEFs são mínimas, completas, determinísticas e inicialmente conexas, conforme apresentado na Figura 5.8.

O Custo médio da geração dos conjuntos de testes dos produtos da LP Mobile Media é representado pela Figura 5.9. Podemos analisar que os conjuntos gerados pelo método HSI possuem o maior custo, enquanto que os conjuntos gerados pela estratégia FSM-TSPL (tanto pelo HSI quanto pelo P) possuem os menores custos. O método P gerou conjuntos com um

Tabela 5.8: Cenários da Mobile Media

Entrega	Descrição	Tipo de Mudança
R1	Núcleo MobilePhoto	
R2	Incluído controle de exceção	4
R3	Adicionado nova característica para contar o número de vezes que a foto é visualizada e ordenando pela freqüência que esta é vista. Nova característica para adicionar/editar rótulo das fotos	1, 3
R4	Nova característica que permite usuários especificar e visualizar suas fotos favoritas	3
R5	Nova característica adicionada que permite usuários manter múltiplas cópias de fotos	3
R6	Nova característica adicionada para enviar foto para outro usuário por SMS	3
R7	Nova característica para armazenar, tocar e organizar músicas. O gerenciamento da foto (criar, apagar e rotular) foi alterado para uma característica alternativa. Todas funcionalidades extendidas (ordenação, favoritos e transferência por SMS) também foram fornecidas	5
R8	Nova característica para gerenciar vídeos	2

Legenda		
	1	Inclusão característica obrigatória
	2	Inclusão característica alternativa
	3	Inclusão característica opcional
	4	Inclusão de interesse não funcional
	5	Mudança de uma característica obrigatória para duas alternativas

Produto	Características
Prod.1	Base + MP
Prod.2	Base + MM
Prod.3	Base + MV
Prod.4	Base + MP + F
Prod.5	Base + MP + CM
Prod.6	Base + MP + SMS
Prod.7	Base + MM + F
Prod.8	Base + MM + CM
Prod.9	Base + MM + SMS
Prod.10	Base + MV + F
Prod.11	Base + MV + CM
Prod.12	Base + MV + SMS
Prod.13	Base + MP + F + CM
Prod.14	Base + MM + F + CM
Prod.15	Base + MV + F + CM
Prod.16	Base + MP + CM + SMS
Prod.17	Base + MM + CM + SMS
Prod.18	Base + MV + CM + SMS
Prod.19	Base + MP + F + SMS
Prod.20	Base + MM + F + SMS
Prod.21	Base + MV + F + SMS
Prod.22	Base + MP + F + CM + SMS
Prod.23	Base + MM + F + CM + SMS
Prod.24	Base + MV + F + CM + SMS

Legenda

MP	Media Phone
MM	Media Music
MV	Media Video
F	Favourites
CM	Copy Media
SMS	SMS Transfer

Figura 5.7: Produtos instanciados da Mobile Media

MEF	Estados	Entradas	Saídas
Prod.1	3	8	2
Prod.2	3	8	2
Prod.3	3	8	2
Prod.4	4	8	2
Prod.5	4	8	2
Prod.6	4	8	2
Prod.7	4	8	2
Prod.8	4	8	2
Prod.9	4	8	2
Prod.10	4	8	2
Prod.11	4	8	2
Prod.12	4	8	2
Prod.13	5	8	2
Prod.14	5	8	2
Prod.15	5	8	2
Prod.16	5	8	2
Prod.17	5	8	2
Prod.18	5	8	2
Prod.19	5	8	2
Prod.20	5	8	2
Prod.21	5	8	2
Prod.22	6	8	2
Prod.23	6	8	2
Prod.24	6	8	2

Figura 5.8: MEFs representando os produtos Mobile Media

custo menor que o HSI, mas maior que os gerados pela FSM-TSPL. A relação entre o custo de geração dos produtos por métodos é ilustrado pela Figura 5.10.

Produto/Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	157	136	–	–
Prod.2	153	106	0	17
Prod.3	153	108	0	67
Prod.4	246	189	104	153
Prod.5	263	206	105	69
Prod.6	245	226	132	179
Prod.7	318	249	178	281
Prod.8	242	156	115	75
Prod.9	264	213	113	109
Prod.10	158	174	117	217
Prod.11	264	186	126	92
Prod.12	238	151	98	101
Prod.13	409	293	172	285
Prod.14	519	342	150	190
Prod.15	269	265	45	66
Prod.16	354	276	110	149
Prod.17	405	253	213	93
Prod.18	269	261	115	155
Prod.19	409	252	198	169
Prod.20	413	307	60	108
Prod.21	363	275	132	27
Prod.22	464	353	287	290
Prod.23	571	415	261	143
Prod.24	474	363	175	192

Figura 5.9: Custo de geração dos conjuntos

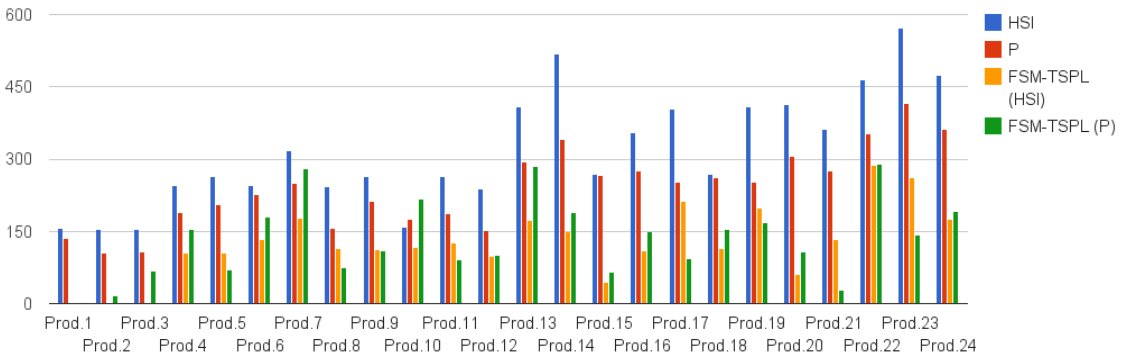

Figura 5.10: Custo da Geração das Seqüências x Produtos/Métodos

O tamanho das operações de reset são apresentados na Figura 5.11. O método HSI também gerou o conjunto de testes com mais operações de resets, da mesma forma que o método P gerou menos operações resets que o HSI. Os conjuntos gerados pela estratégia FSM-TSPL possuem um menor número de operações reset comparada com os outros métodos, conforme Figura 5.12.

Produto/Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	37	30	–	–
Prod.2	36	22	0	3
Prod.3	36	22	0	10
Prod.4	51	37	18	26
Prod.5	58	44	21	14
Prod.6	51	44	22	32
Prod.7	66	50	28	49
Prod.8	50	29	19	11
Prod.9	58	42	21	21
Prod.10	36	29	18	34
Prod.11	58	37	23	17
Prod.12	50	29	15	18
Prod.13	79	53	23	41
Prod.14	102	62	22	31
Prod.15	58	46	9	11
Prod.16	72	53	22	25
Prod.17	78	42	31	16
Prod.18	58	44	20	25
Prod.19	79	45	32	28
Prod.20	80	56	9	18
Prod.21	71	40	20	4
Prod.22	87	62	45	42
Prod.23	109	70	40	20
Prod.24	93	56	26	25

Figura 5.11: Quantidade de operações reset

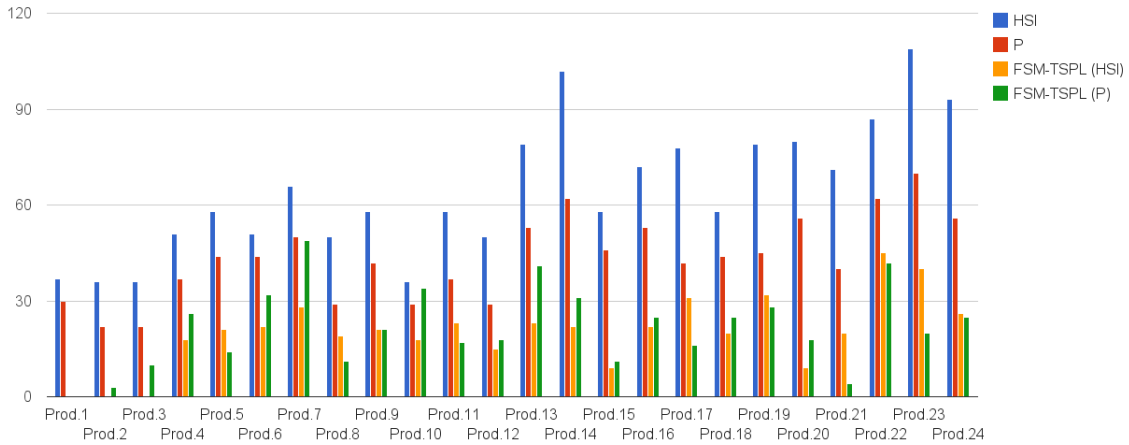

Figura 5.12: Número Resets x Produtos/Métodos

O Comprimento das seqüências geradas pelos métodos são apresentadas na Figura 5.13. Conforme podemos analisar, o método HSI gera as menores seqüências, enquanto que o gerados pela FSM-TSPL possuem tamanho igual ou maior, esta relação é ilustrada pela Figura 5.14.

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	3	3	–	–
Prod.2	3	3	0	4
Prod.3	3	3	0	5
Prod.4	3	4	4	4
Prod.5	3	3	4	3
Prod.6	3	4	5	4
Prod.7	3	3	5	4
Prod.8	3	4	5	5
Prod.9	3	4	4	4
Prod.10	3	5	5	5
Prod.11	3	4	4	4
Prod.12	3	4	5	4
Prod.13	4	4	6	5
Prod.14	4	4	5	5
Prod.15	3	4	4	5
Prod.16	3	4	4	4
Prod.17	4	5	5	4
Prod.18	3	4	4	5
Prod.19	4	4	5	5
Prod.20	4	4	5	5
Prod.21	4	5	5	4
Prod.22	4	4	5	5
Prod.23	4	4	5	6
Prod.24	4	5	5	6

Figura 5.13: Comprimento das seqüências

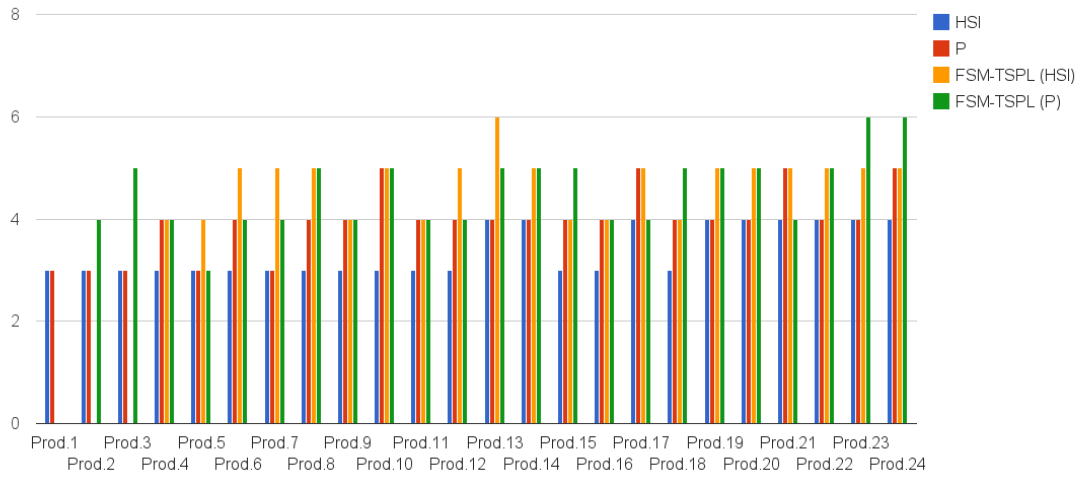

Figura 5.14: Comprimento Médio das Seqüências x Produtos/Métodos

O tempo médio de execução dos algoritmos para geração dos conjuntos de testes é apresentado pela Figura 5.15. Da mesma forma que no experimento 1, o algoritmo do HSI levou menos tempo para gerar os conjuntos de testes, seguido do P e a FSM-TSPL levou mais tempo que os outros métodos, conforme ilustrado pela Figura 5.16.

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	27	3670	–	–
Prod.2	28	3530	4556	5403
Prod.3	28	3700	4216	7894
Prod.4	51	16321	15959	49651
Prod.5	60	20460	124915	78049
Prod.6	58	24869	146886	159180
Prod.7	75	43638	171198	597089
Prod.8	52	14256	152098	148574
Prod.9	61	32327	120673	97309
Prod.10	32	27652	113314	373361
Prod.11	61	24678	46699	134612
Prod.12	52	11326	116022	94895
Prod.13	98	127759	247686	752115
Prod.14	155	251159	813745	947387
Prod.15	64	115695	594866	411404
Prod.16	102	78098	188363	424805
Prod.17	101	87866	442940	186412
Prod.18	71	127774	438417	365638
Prod.19	99	63288	262030	629587
Prod.20	100	129195	381733	285087
Prod.21	85	208517	441656	195539
Prod.22	120	190108	993603	2059780
Prod.23	175	494526	2305216	1425227
Prod.24	136	541143	1827950	3253824

Figura 5.15: Tempo médio de execução dos algoritmos em *ms*

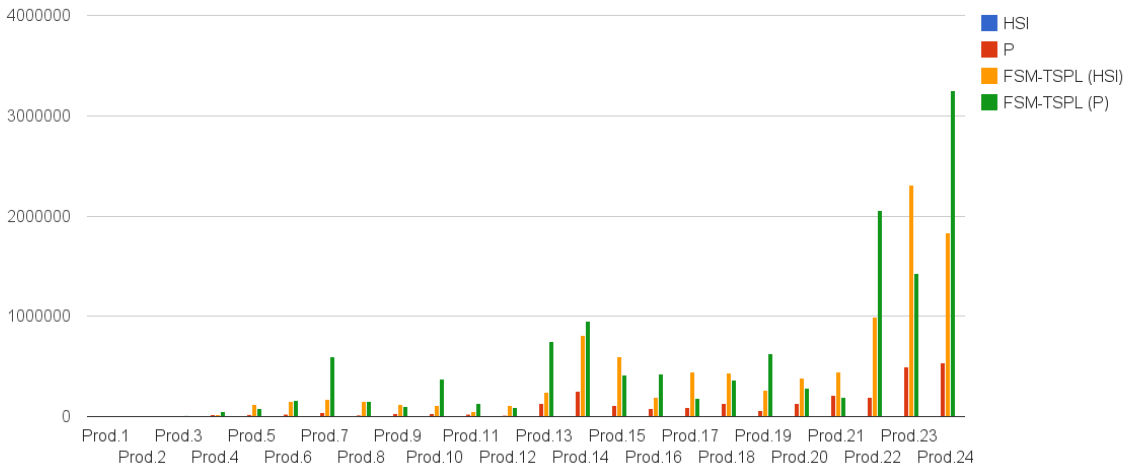

Figura 5.16: Tempo médio de execução dos algoritmos em *ms*

A Figura 5.17 apresenta os custos acumulativos para testar todos os produtos da LP Mobile Media. Podemos observar que o método HSI possui o maior custo, o método P possui um custo médio e similar, enquanto o FSM-TSPL possui o menor custo, de maneira que o FSM-TSPL (HSI) possui o custo total de *3006*. A relação dos custos acumulativos por método é ilustrada pela Figura 5.18.

Método	HSI	P	FSM-TSPL (HSI)	FSM-TSPL (P)
Prod.1	157	136	–	–
Prod.[1-2]	310	242	0	17
Prod.[1-3]	463	350	0	84
Prod.[1-4]	709	539	104	237
Prod.[1-5]	972	745	209	306
Prod.[1-6]	1217	971	341	485
Prod.[1-7]	1535	1220	519	766
Prod.[1-8]	1777	1376	634	841
Prod.[1-9]	2041	1589	747	950
Prod.[1-10]	2199	1763	864	1167
Prod.[1-11]	2463	1949	990	1259
Prod.[1-12]	2701	2100	1088	1360
Prod.[1-13]	3110	2393	1260	1645
Prod.[1-14]	3629	2735	1410	1835
Prod.[1-15]	3898	3000	1455	1901
Prod.[1-16]	4252	3276	1565	2050
Prod.[1-17]	4657	3529	1778	2143
Prod.[1-18]	4926	3790	1893	2298
Prod.[1-19]	5335	4042	2091	2467
Prod.[1-20]	5748	4349	2151	2575
Prod.[1-21]	6111	4624	2283	2602
Prod.[1-22]	6575	4977	2570	2892
Prod.[1-23]	7146	5392	2831	3035
Prod.[1-24]	7620	5755	3006	3227

Figura 5.17: Custo acumulativo dos conjuntos de teste

Por meio da utilização da FSM-TSPL (HSI) gerou uma economia de *61%* comporado HSI e *48%* em relação ao P. Os resultados da utilização da FSM-TSPL (P) geraram menos economia sendo estas *58%* para o HSI e *44%* para o P.

5.4 Considerações Finais

Ao longo deste Capítulo foram conduzidos dois experimentos, AGM e Mobile Media, de maneira a avaliar a estratégia FSM-TSPL. Em ambos os experimentos a estratégia mostrou-se bastante efetiva na geração e reutilização de conjuntos de testes entre os produtos, gerando os conjuntos com menores custos e operações resets se comparada com outros métodos.

No capítulo seguinte são apresentadas as conclusões deste trabalho de mestrado, com as contribuições, dificuldades, limitações e direções para trabalhos futuros.

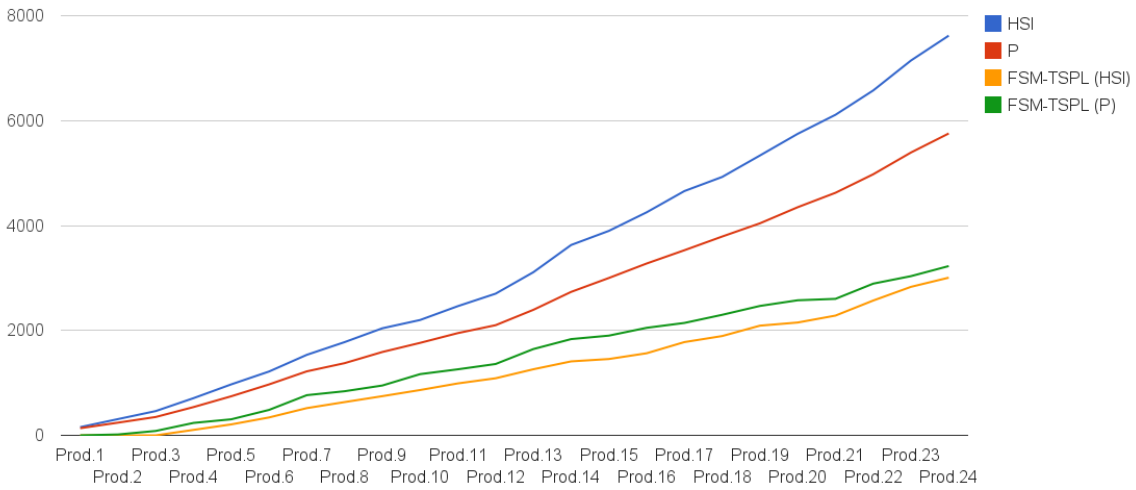

Figura 5.18: Custo Acumulativo da Geração das Seqüências x Produtos/Métodos

Capítulo

6

Conclusões

Teste em LP não é um processo trivial, devido ao grande número de configurações de produtos, derivados a partir das variabilidades. Os testes são divididos em duas etapas: engenharia de domínio e engenharia de aplicação. Apesar de existirem diversas abordagens de teste em LP, uma das dificuldades encontradas é a reutilização de teste entre os produtos instanciados da LP. Neste trabalho de mestrado, foi investigado este problema e elaborada uma estratégia de geração de testes incremental chamada de FSM-TSPL. Esta abordagem reutiliza testes gerados a partir de uma MEF M_1, que representa o produto da LP $prod_1$, para testar uma MEF M_2 representando o produto $prod_2$. Os conjuntos de testes servem como entrada para o algoritmo P, que o incrementa até que o mesmo satisfaça a MEF M_2, garantindo sua conformidade.

Foram realizados dois experimentos: AGM o qual representa uma LP de jogos para dispositivos móveis; e Mobile Media que representa uma LP contendo diversas funcionalidades de multimídia também para dispositivos móveis.

A seguir são apresentadas as principais contribuições deste trabalho de mestrado, assim como as dificuldades, limitações e perspectivas de trabalhos futuros.

6.1 Contribuições

A principal contribuição deste trabalho de mestrado foi a definição de uma estratégia de testes incremental chamada de FSM-TSPL. Esta abordagem utiliza MEFs como modelos dos produtos e os conjuntos de testes são as seqüências de testes geradas por métodos de geração

de casos de testes baseados em MEF, tais como: W, HSI, P, entre outros. Na abordagem utilizamos apenas dois métodos de geração sendo estes: HSI e P. No lugar do método HSI poderia ser utilizado qualquer outro método e/ou até mesmo utilizar testes *ad hoc*.

Foram realizados dois experimentos, usando duas LPs, em que a abordagem foi utilizada para gerar o conjunto de testes e estes reutilizados entre os produtos de cada LP. Para avaliar a utilização da estratégia FSM-TSPL foram realizadas algumas análises comparativas avaliando o custo de geração do conjunto de testes, quantidade de operações, tamanho das seqüências, tempo de execução e custo acumulativo, utilizando o método HSI, P, a estratégia FSM-TSPL contendo o conjunto de testes gerado pelo HSI como entrada e outra FSM-TSPL contendo os conjuntos do método P.

Foram implementados algoritmos HSI, na linguagem Ruby tanto para a interpretação de MEFs, que aborda duas DSLs KISS e RFSM, quanto para a comparação entre casos de testes (custos, tamanho seqüências e operações reset) e geradores de grafos (MEF, árvore de testes e grafo de distinção).

6.2 Limitações e Trabalhos Futuros

Na Seção 5.3, foi evidenciado que a estratégia FSM-TSPL (P) obteve um custo maior que as outras abordagens para realizar o teste no produto $prod_{10}$. Desta forma, uma análise mais detalhada buscando entender este problema se faz necessária. Quanto a escolha da ordem da execução dos testes entre os produtos, poderia haver um estudo mais detalhado apontando qual seria a melhor ordem e se estas escolhas influenciam no resultado final.

A geração dos conjuntos de testes incremental pelo algoritmo P pode haver melhorias, de maneira a diminuir o tamanho dos conjuntos de testes. Ainda no método P, poderia ser implementado uma funcionalidade para refinar o conjunto de testes quando é utilizado um conjunto de teste como entrada.

Outro ponto a se investigar seria a utilização de MEFs parciais na representação dos produtos e na geração de testes para estes, de forma a melhor representar as variabilidades pertinentes a LPs.

Referências

Apfelbaum, L.; Doyle, J., Model based testing. In: *Software Quality Week Conference*, 1997, p. 296–300.

Barroca, L. M.; Mcdermid, J. A., Formal methods: Use and relevance for the development of safety critical systems. *The Computer Journal*, v. 35, p. 579–599, 1992.

Bayer, J.; Flege, O.; Knauber, P.; Laqua, R.; Muthig, D.; Schmid, K.; Widen, T.; DeBaud, J.-M., Pulse: a methodology to develop software product lines. In: *SSR '99: Proceedings of the 1999 symposium on Software reusability*, New York, NY, USA: ACM, 1999, p. 122–131.

Bertolino, A.; Gnesi, S., Use case-based testing of product lines. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, New York, NY, USA: ACM, 2003, p. 355–358 (*ESEC/FSE-11*,).
Disponível em `http://doi.acm.org/10.1145/940071.940120`

Broy, M.; Jonsson, B.; Katoen, J.-P.; Leucker, M.; Pretschner, A., *Model-based testing of reactive systems: advanced lectures*. 1st ed. Springer, 2005.

Cechticky, V.; Pasetti, A.; Rohlik, O.; Schaufelberger, W., Xml-based feature modelling. *Proceedings of ICSR*, p. 101–114, 2004.

Chow, T. S., Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, v. 4, p. 178–187, 1978.
Disponível em `http://portal.acm.org/citation.cfm?id=1313335.1313730`

Clements, P.; Northrop, L., *Software product lines: Practices and patterns*. Addison-Wesley Longman Publishing Co., 2001.

Czarnecki, K.; Helsen, S.; Eisenecker, U., Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, v. 10, p. 143–169, 2005.
Disponível em `http://dx.doi.org/10.1002/spip.225`

Dalal, S. R.; Jain, A.; Karunanithi, N.; Leaton, J. M.; Lott, C. M.; Patton, G. C.; Horowitz, B. M., Model-based testing in practice. In: *Proceedings of the 21st international conference on Software engineering*, ICSE '99, New York, NY, USA: ACM, 1999, p. 285–294 (*ICSE '99*,).
Disponível em `http://doi.acm.org/10.1145/302405.302640`

Delamaro, M. E.; Maldonado, J. C.; Jino, M., *Introdução ao teste de software.* Elsevier Editora Ltda, 2007.

Demillo, R. A., Mutation analysis as a tool for software quality assurance. In: *COMPSAC80*, Chicago, IL, 1980, p. 390–393.

van Deursen, A.; Klint, P., Little languages: little maintenance. *Journal of Software Maintenance*, v. 10, p. 75–92, 1998.
Disponível em `http://dl.acm.org/citation.cfm?id=278004.278005`

van Deursen, A.; Klint, P.; Visser, J., Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, v. 35, p. 26–36, 2000.
Disponível em `http://doi.acm.org/10.1145/352029.352035`

Dorofeeva, R.; El-Fakih, K.; Yevtushenko, N., An improved conformance testing method. In: Wang, F., ed. *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, v. 3731 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 204–218, 10.1007/11562436_16, 2005.
Disponível em `http://dx.doi.org/10.1007/11562436_16`

Figueiredo, E.; Cacho, N.; SantÁnna, C.; Monteiro, M.; Kulesza, U.; Garcia, A.; Soares, S.; Ferrari, F.; Khan, S.; Castor Filho, F.; Dantas, F., Evolving software product lines with aspects: an empirical study on design stability. In: *Proceedings of the 30th international conference on Software engineering*, ICSE '08, New York, NY, USA: ACM, 2008, p. 261–270 (*ICSE '08*,).
Disponível em `http://doi.acm.org/10.1145/1368088.1368124`

Fowler, M., *Domain-specific languages.* Addison-Wesley Professional, 2010.

Fujiwara, S.; von Bochmann, G.; Khendek, F.; Amalou, M.; Ghedamsi, A., Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, v. 17, p. 591–603, 1991.
Disponível em `http://portal.acm.org/citation.cfm?id=126218.126234`

Furbach, U., Formal specification methods for reactive systems. *J. Syst. Softw.*, v. 21, p. 129–139, 1993.
Disponível em `http://dl.acm.org/citation.cfm?id=153678.153682`

Gill, A., *Introduction to the theory of finite-state machines.* McGraw-Hill, 1962.

Gimenes, I. M. S.; Travassos, G. H., O enfoque de linha de produto para desenvolvimento de software. Anais da XXI Jornada de Atualização em Informática (JAI) - SBC, 2002.

Goldstein, S., *Writing domain specific languages (dsls) with ruby.* 2009.
Disponível em `http://drasticcode.com/2009/8/3/writing-domain-specific-languages-dsls-with-ruby`

Gomaa, H., *Designing software product lines with uml: From use cases to pattern-based software architectures.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

GOMAA, H., Designing software product lines with uml. In: *SEW '05: Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop - Tutorial Notes*, Washington, DC, USA: IEEE Computer Society, 2005, p. 160–216.

GONENC, G., A method for the design of fault detection experiments. *IEEE Trans. Comput.*, v. 19, p. 551–558, 1970.
Disponível em `http://dx.doi.org/10.1109/T-C.1970.222975`

HAREL, D.; PINNEL, A. S. J. S. R., On the fornal semantics of statecharts. In: *2nd IEEE Symposium on Logic in Computer Science*, New York: IEEE Press, 1987, p. 54–64.

HEYMANS, P.; TRIGAUX, J. C., *Software product line: state of the art.* Relatório Técnico, Institut d'Informatique FUNDP, Manur, 2003.

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LÜTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H., Using formal specifications to support testing. *ACM Comput. Surv.*, v. 41, p. 9:1–9:76, 2009.
Disponível em `http://doi.acm.org/10.1145/1459352.1459354`

HIERONS, R. M.; URAL, H., Generating a checking sequence with a minimum number of reset transitions. *Automated Software Engg.*, v. 17, p. 217–250, 2010.
Disponível em `http://dx.doi.org/10.1007/s10515-009-0061-0`

IEEE, *Ieee standard glossary of software engineering terminology.* Relatório Técnico, 1990.
Disponível em `http://dx.doi.org/10.1109/IEEESTD.1990.101064`

IM, K.; IM, T.; MCGREGOR, J. D., Automating test case definition using a domain specific language, p. 180–185. 2008.

JIN-HUA, L.; QIONG, L.; JING, L., The w-model for testing software product lines. In: *ISCSCT '08: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology*, Washington, DC, USA: IEEE Computer Society, 2008, p. 690–693.

KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S., *Feature-oriented domain analysis (foda) feasibility study.* Relatório Técnico, Carnegie-Mellon University Software Engineering Institute, 1990.

VAN DER LINDEN, F. J.; SCHMID, K.; ROMMES, B., *Software product lines in action: The best industrial practice in product line engineering.* Springer, 2007.

LUO, G.; PETRENKO, A.; PETRENKO, R.; BOCHMANN, G. V., Selecting test sequences for partially-specified nondeterministic finite state machines. In: *In IFIP 7th International Workshop on Protocol Test Systems*, 1994, p. 91–106.

MALDONADO, J. C., *Critério potenciais usos: Uma contribuićão ao teste estrutural de software.* Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, 1991.

MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M., *Introdução ao teste de software.* Relatório Técnico, ICMC/USP, São Carlos, SP, 2004.

McGregor, J. D., *Testing a software product line.* Relatório Técnico, Software Engineering Institute (SEI), Pittsburgh, PA, 2001.
Disponível em `http://www.sei.cmu.edu/reports/01tr022.pdf`

MIT, S. D. G., *Alloy.* 1997.
Disponível em `http://alloy.mit.edu/faq.php`

Myers, G. J.; Sandler, C., *The art of software testing.* John Wiley & Sons, 2004.

Olimpiew, E. M.; Gomaa, H., Reusable model-based testing. In: *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, Berlin, Heidelberg: Springer-Verlag, 2009, p. 76–85.

de Oliveira Júnior, E. A., *System-pla : um método sistemático para avaliaćão de arquitetura de linha de produto de software baseada em uml.* Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computaćão - ICMC, Universidade de São Paulo, São Carlos, 2010.

Petrenko, A.; Yevtushenko, N.; Lebedev, A.; Das, A., Nondeterministic state machines in protocol conformance testing. In: *Proceedings of the IFIP TC6 WG6.1 Sixth International Workshop on Protocol Test systems VI*, Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1994, p. 363–378.
Disponível em `http://dl.acm.org/citation.cfm?id=648128.761244`

Petrenko, R.; Yevtushenko, N., Testing from partial deterministic fsm specifications. *IEEE Transactions on Computers*, p. 1154–1165, 2005.

Petri, C. A., *Kommunikation mit automaten.* Tese de Doutoramento, Institut für instrumentelle Mathematik, Bonn, 1962.

Pohl, K.; Böckle, G.; van der Linden, F. J., *Software product line engineering: Foundations, principles and techniques.* Springer-Verlag, 2005.

Pohl, K.; Metzger, A., Software product line testing. *Commun. ACM*, v. 49, n. 12, p. 78–81, 2006.

Pressman, R., *Software engineering: A practitioners approach.* 6 ed. New York, NY, USA: McGraw-Hill, Inc., 2005.

Sabnani, K.; Dahbura, A., A protocol test generation procedure. *Comput. Netw. ISDN Syst.*, v. 15, p. 285–297, 1988.
Disponível em `http://dl.acm.org/citation.cfm?id=48478.48482`

SEI, A framework for software product line practice. 2011.
Disponível em `http://www.sei.cmu.edu/productlines/framework.html`

Simão, A.; Petrenko, A., Fault Coverage-Driven Incremental Test Generation. *The Computer Journal*, v. 53, n. 9, p. 1508–1522, 2010.
Disponível em `http://comjnl.oxfordjournals.org/content/53/9/1508.abstract`

Tevanlinna, A.; Taina, J.; Kauppinen, R., Product family testing: a survey. *SIGSOFT Softw. Eng. Notes*, v. 29, n. 2, p. 12–12, 2004.

Uzuncaova, E.; Garcia, D.; Khurshid, S.; Batory, D., A specification-based approach to testing software product lines. In: *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA: ACM, 2007, p. 525–528.

Uzuncaova, E.; Garcia, D.; Khurshid, S.; Batory, D., Testing software product lines using incremental test generation. In: *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, Washington, DC, USA: IEEE Computer Society, 2008, p. 249–258.

Vuong, S.T., C. W. I. M., The uiov-method for protocol test sequence generation. In: *Proceedings of the 2nd International Workshop on Protocol Test Systems*, 1989.

Weiss, D. M.; Lai, C. T. R., *Software product-line engineering: a family-based software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

Young, T. J.; Young, T. J., Using aspectj to build a software product line for mobile devices. msc dissertation. In: *University of British Columbia, Department of Computer Science*, 2005, p. 1–6.