UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

ISSN 0103-2569

Módulo para geração de casos de teste a partir de Máquinas de Estados Finitos Parciais

Jorge Francisco Cutigi

Adenilso da Silva Simão

Nº 286

RELATÓRIOS TÉCNICOS

São Carlos - SP

Dezembro/2006

Resumo

Grande parte dos métodos de geração de seqüências existentes tratam apenas de Máquinas de Estados Finitos totalmente especificadas, o que torna o uso restrito já que, na maioria das situações, as Máquinas de Estados Finitos testadas não apresentam essa característica. Este relatório apresenta o processo de desenvolvimento de um módulo que implementa um algoritmo para a geração de casos de teste a partir de Máquinas de Estados Finitos Parciais. O algoritmo estudado foi o Método HSI, sendo implementado e integrado na ferramenta Plavis — Platform for Software Validation & Integration on Space Systems —, uma ferramenta de apoio a verificação, validação e teste de software.

Sumário

1 Introdução

O uso de software nas mais diversas áreas de aplicação impõe a necessidade de técnicas e ferramentas que auxiliam seu desenvolvimento. Entre as etapas do desenvolvimento de software, encontra-se a fase de teste, a qual deve ser feita com apoio de técnicas, métodos e ferramentas precisas e eficazes, uma vez que essa fase é crucial para se obter um software confiável e de boa qualidade.

Existem várias técnicas de teste de software, dentre as quais, pode-se citar a técnica funcional, a técnica estrutural, a técnica baseadas em erros e a técnica baseada em máquinas de estados finitos [12]. A diferença básica entre essas técnicas é a origem da informação que é utilizada para derivar os casos de teste. A técnica baseada em máquina de estados finitos utiliza-se de uma máquina de estados finitos (MEF) para modelar o sistema em teste. Tendo criada a MEF, o teste caracteriza-se por um conjunto de entradas que, quando executadas, geram as saídas esperadas. Essas saídas são confrontadas com as saídas obtidas pelo sistema em teste. Existem vários métodos de geração de seqüências de teste. O conjunto de seqüências de teste deve apresentar determinadas características para que sua execução seja viável, tais como número pequeno de seqüências, fácil e rápida execução e máxima identificação de falhas de um sistema. Dentre os métodos mais conhecidos podemos destacar o método W [3], DS [10], UIO [15, 1], Wp[8], Switch-Cover [13], HSI [11, 17], H [18] e State Counting [14].

O projeto Plavis [16] objetiva estabelecer um trabalho em conjunto com institutos que possuem o domínio da tecnologia em pesquisas, de forma a obter a melhor prática para o apoio a atividades de VV&T — Verificação, Validação e Teste. Objetiva-se desenvolver uma plataforma integre vários módulos de VV&T que atualmente são utilizados pelo Instituto Nacional de Pesquisas Espaciais (INPE) para geração de estudos de caso dos softwares espaciais. Atualmente, a plataforma possui módulos para a geração de casos de teste a partir de MEFs com os métodos W, UIO e Switch-Cover. Para permitir que estudos comparativos sejam realizados, é importante que novos módulos sejam implementados para apoiar a geração de casos de teste para os métodos encontrados na literatura.

Neste trabalho, foi proposto o estudo do método HSI que pode resolver o problema de MEFs parciais, oferecendo uma alternativa ao método W. Após o estudo e implementação de um módulo para geração de casos de teste para MEFs parciais, este será integrado na ferramenta PLAVIS/FSM, cujo objetivo é oferecer apoio a verificação, validação e testes de MEFs.

Este documento está organizado da seguinte forma. Na Seção 2 serão apresentados o conceito e a representação de MEFs, assim como propriedades, que são necessárias para alguns métodos de geração de casos de teste. Na Seção 3 será apresentado o método HSI, ilustrando suas características e o processo de geração de casos de teste, além de um exemplo da sua aplicação. Na Seção 4 serão apresentados os aspectos de implementação e computacionais do módulo desenvolvido. Na Seção 5 será apresentada uma descrição sobre o projeto Plavis e dobre a integração do módulo de geração de casos de teste. Na Seção 6 serão apresentadas as dificuldades e limitações ocorridas durante o desenvolvimento do projeto Na Seção 7 serão apresentadas as conclusões e os objetivos alcançados.

2 Máquinas de Estados Finitos

Uma MEF [9] é uma máquina hipotética composta por estados e transições. Cada transição liga um estado a a um estado b (a e b podem ser o mesmo estado). A cada instante, uma máquina pode estar em apenas um de seus estados. Em resposta a um evento de entrada, a máquina gera um evento de saída e muda de estado. Tanto o evento de saída gerado quanto o novo estado são definidos unicamente em função do estado atual e do evento de entrada [4].

De acordo com Fujiwara [8], uma MEF pode ser representada por uma tupla $M = (X, S, S_0, D, Y, T, O)$, onde:

- X é o conjunto de entradas;
- S é o conjunto de estados, incluindo um estado especial S_0 chamado de estado inicial;
- D é o conjunto dos pares estado/entrada que a MEF aceita, onde $D \subseteq S \times X$;
- Y é o conjunto de saídas, incluindo uma saída nula;
- T é a função de transição, $D \rightarrow S$;
- O é a função de saída, $D \rightarrow Y$.

Essas características representam MEFs parciais, porém se $D \rightarrow S \times X$ então a MEF é completamente especificada.

MEFs podem ser representadas graficamente, de acordo com uma tabela de transição de estados (Tabela 1) ou por um diagrama de transição de estados (Figura 1).

Um exemplo de leitura da MEF da Figura 1 e da Tabela 1, seria que o estado *2* com entrada *a*, realiza a transição para o estado *3*, produzindo saída *1*, já o estado *3* com entrada *b*, não realiza nenhuma transição de estados, já que a entrada *b* não está definida nesse estado e, conseqüentemente, não produz saída.

	Y		E	
Estado	a	b	a	b
1	1	0	2	3
2	1	0	3	4
3	0	-	4	-
4	-	0	-	1

Tabela 1: Representação da MEF como uma tabela de transição

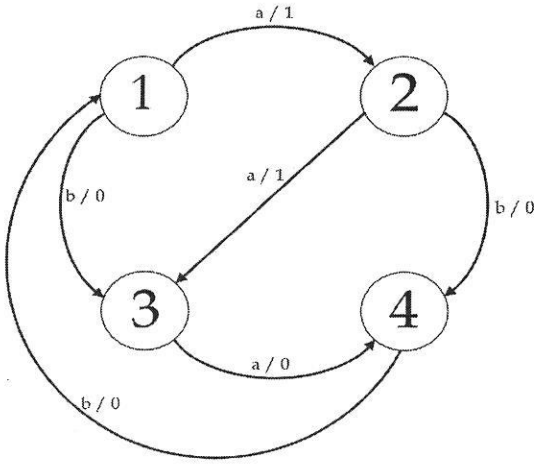

Figura 1: Representação da MEF da Tabela 1 como diagrama de transição de estados

Diversas propriedades podem ser verificadas em uma MEF, sendo que alguns métodos de geração necessitam que tais propriedades sejam satisfeitas. Dentre as propriedades pode-se citar:

Completamente especificada: para todo estado $s \in S$ deve existir uma transição para cada símbolo de entrada $x \in X$, ou seja, $D = S \times X$;

Minimal: se nenhum par de estados da máquina é equivalente; Ou seja, para todo par de estados (G_i, G_j), existe ao menos uma seqüência de entradas cuja saída produzida seja distinta;

Determinística: para cada entrada existe no máximo uma transição definida em cada estado da máquina;

Máquina de Mealy: toda transição deve conter uma saída;

Fortemente conectada: para todo par de estados (G_i, G_j) da máquina, existe uma seqüência de símbolos de entrada que leve de G_i até G_j.

3 Método HSI

As propriedades apresentadas na Seção 2 geralmente não são satisfeitas pela maioria das MEFs. Dessa forma, há a necessidade de métodos para atender as MEFs parciais, que fogem às propriedades *Completamente Especificada* e *Máquina de Mealy*. Uma solução para esse problema seria completar artificialmente a MEF, adicionando arcos nulos nos estados não totalmente especificados. Com isso, entretanto, tem-se o problema de um grande número de casos de teste e os arcos artificiais poderiam interferir no cálculo dos casos de teste, gerando, na maioria dos casos, um conjunto de teste inviável.

Nesse contexto encontra-se o método HSI (*Harmonized State Identification*) [11, 17] que resolve o problema das MEFs parciais. O Método HSI é similar ao método W [3], diferenciando no critério de seleção do conjunto de caracterização, selecionando um subconjunto deste, chamado de conjunto HSI. Por ser menor que o conjunto W, o conjunto de seqüências de teste final também será menor, o que otimiza o processo de teste final. Com essa diferenciação na seleção do conjunto de caracterização, o método HSI tem a capacidade de gerar casos de testes que antes não eram possíveis com outros métodos.

A seguir, é apresentada uma descrição informal do método HSI. Os algoritmos utilizados são descritos na Seção 4.

Para se obter o conjunto final de testes a partir do método HSI, precisa-se dos seguintes conjuntos:

Preâmbulo: esse conjunto caracteriza-se, por ser a sequência de entradas que leva a MEF do estado inicial para determinado estado;

Transition Cover: conjunto de sequências que inclui o preâmbulo de cada estado, concatenado com as entradas aceitas pelo estado atual;

Identificador de estado: se trata do conjunto HSI, que se caracteriza por identificar o estado atual da MEF.

3.1 Obtenção do Preâmbulo e Transition Cover

Para se obter o *preâmbulo* de uma MEF, cria-se uma árvore, na qual a raiz da árvore é o estado inicial da MEF, sendo que a partir dessa raiz cria-se a árvore, parando quando

todos os estados da MEF já estiverem na árvore. A Figura 2 ilustra a árvore de obtenção do *preâmbulo* para a MEF da Figura 1.

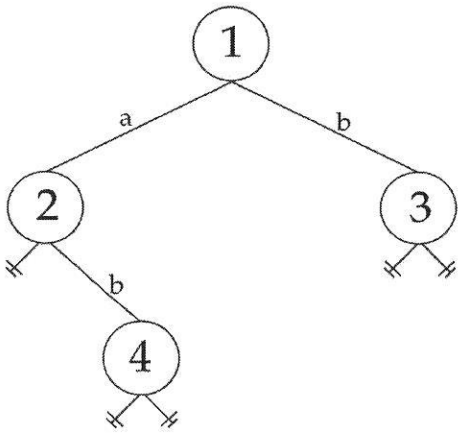

Figura 2: Árvore utilizada para se obter o preâmbulo.

Percorrendo-se a árvore, tem-se o *preâmbulo* de cada estado, e concatenando-se esse conjunto com as entradas aceitas pelo estado atual, obtem-se o *Transition Cover*. Na Tabela 2 pode ser visualizado o *preâmbulo* e o *Transition Cover* para a MEF da Figura 1

Estado	1	2	3	4
Preâmbulo	-	a	b	ab
Transition Cover	a, b	aa, ab	ba	abb

Tabela 2: Preâmbulo e Transition Cover de cada estado da MEF da Figura 1.

3.2 Obtenção do Conjunto HSI

Depois de encontrados os conjuntos necessários, aplica-se um algoritmo para encontrar o conjunto identificador de estados, ou seja, o conjunto HSI. O algoritmo pode ser dividido em três passos.

3.2.1 Passo 1

O primeiro passo, A_1, consiste em construir uma lista L_0 com todos os pares de estados da MEF. A Figura 3 ilusta o passo A_1 para a MEF da Figura 1.

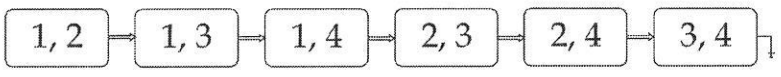

Figura 3: Lista L_0.

3.2.2 Passo 2

No segundo passo, A_2, para os dois estados de cada nó da lista L_0, aplicam-se as entradas aceitas pela MEF. Dessa forma, os dois estados do nó de L_0 realizarão uma transição de estados, formando assim uma outra dupla de estados. Se essa dupla de estados for identificada em L_0, então cada entrada da MEF para o nó de L_0 aponta para a dupla de estados identificada. Para a MEF da Figura 1, o primeiro nó de L_0 (estados 1 e 2), com entrada a, realiza transição para o estado 2 e 3, respectivamente. Já com entrada b, realiza transição para o estado 3 e 4. Sendo assim a entrada a em L_0 aponta para o quarto nó de L_0 (estados 2 e 3) e a entrada b aponta para sexto nó de L_0 (estados 3 e 4).

Se para algum estado de um nó L_0, a entrada executada não estiver definida, então o ponteiro das entradas de L_0 aponta para *null*. Como exemplo temos o terceiro nó de L_0 (estados 2 e 3), sendo que a entrada b não está definida para o estado 3.

Na Figura 4 temos um exemplo completo do passo A_2, utilizando a MEF da Figura 1.

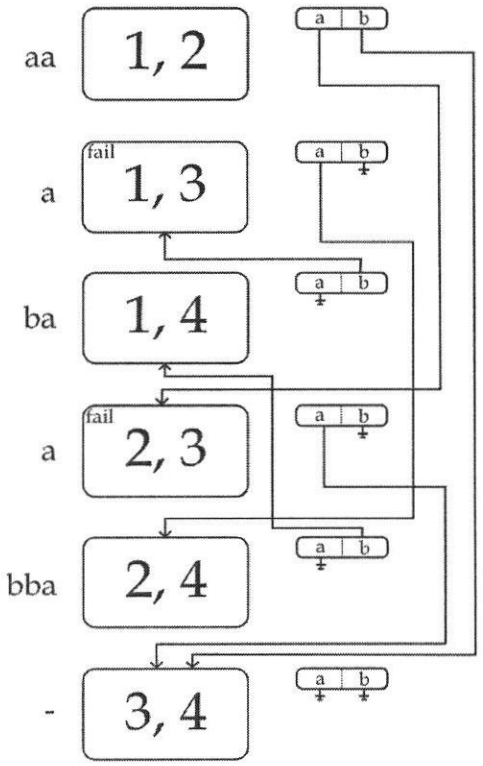

Figura 4: Estrutura utilizada no passo A_2.

3.2.3 Passo 3

No terceiro e último passo A_3, é marcada em cada nó de L_0 a entrada que é aceita por qualquer um dos estados de cada nó de L_0. Se essa entrada gerar saídas diferentes para

os dois estados, o nó é marcado com uma flag, indicando um par de estados $fail$. Tem-se como exemplo o quarto nó de L_0 (estados 2 e 4), onde a entrada b, que é definida por ambos os estados, gera saídas diferente, indicando assim um estado $fail$, onde também é marcado com a entrada b.

Para os nós de L_0 que não são $fail$, completa-se a entrada identificadora desse nó, com a entrada aceita pelo mesmo, concatenada com a sequência de entradas que leva esse nó a um outro nó $fail$. Tem-se como exemplo o terceiro nó de L_0 (estados 1 e 4), onde a entrada aceita é b, porém este aponta para o segundo nó de L_0 (estados 1 e 3), que é $fail$, portanto a entrada identificadora se dá pela concatenação de b com a entrada a (nó $fail$), resultando na entrada identificadora ba. O passo a_3 também pode ser visualizado na Figura 4.

Ao fim, para cada estado da MEF, é selecionado o conjunto identificador de estados. Esse conjunto é obtido verificando-se cada estado da MEF contido em L_0, adicionando ao conjunto identificador desse estado, todas as entradas relacionadas a ele encontradas em A_3. Tem-se como exemplo o estado 2, onde as entradas em A_3 que estão relacionadas com este estado são {aa, a, bba}, sendo o conjunto identificador final (excluindo os prefixos) o conjunto {aa, bba}

Para o exemplo da MEF da Figura 1 o conjunto HSI ficaria:

$$HSI = \left\{ \begin{array}{l} HSI_1 = \{\texttt{aa, ba}\} \\ HSI_2 = \{\texttt{aa, bba}\} \\ HSI_3 = \{\texttt{a}\} \\ HSI_4 = \{\texttt{ba, bba}\} \end{array} \right\}$$

3.3 Obtenção do Conjunto de Teste

Para se obter o conjunto de teste final, executa-se o *Transition Cover*, verificando em qual estado a execução terminou, e então concatena-se a esse *Transition Cover* o HSI do estado atual. Exemplificando, tem-se o *Transition Cover* {ab}. Aplicando-se esse conjunto de entradas na MEF da Figura 1, termina-se a execução no estado 4, sendo assim, obtemos o conjunto para esse *Transition Cover* concatenando-se o estado identificador do estado 4, resultando no conjunto de sequências {abba, abbba}.

Assim, o conjunto de teste final da MEF da Figura 1 pode ser visto abaixo:

Conjunto Final = {aaa, abba, ba, aaa, abba, abbba, baba, babba, abbaa, abbba}

4 Aspectos de Implementação

Nas sub-seções abaixo, são explicados os aspectos de implementação das estruturas de dados utilizados. Na Sub-Seção 4.1 serão apresentadas a estruturas de dados utilizadas para representar as MEFs. Na Sub-Seção 4.2 serão apresentadas as estruturas utilizadas e como é realizada a obtenção do *Preâmbulo* e do *Transition Cover* através dessa estrutura. Na Sub-Seção 4.3 serão apresentadas as estruturas de dados, e como é feito a obtenção do *conjunto HSI*. Na Sub-Seção **??** será apresentado o algoritmo de obtenção do preâmbulo. Na Sub-Seção 4.5 será apresentado o algoritmo para obtenção do conjunto HSI. Na Sub-Seção 4.6 será apresentado o algoritmo para obtenção do conjunto final de casos de teste.

Todos os códigos das estruturas dessa seção estão na linguagem C.

4.1 Estrutura para Representação de Máquinas de Estados Finitos

Como vimos, uma MEF pode ser representada por uma tabela de transição de estados. Essa tabela é então representada por uma matriz, conforme estrutura abaixo:

```
typedef struct {
   char estado[MAX2];
   char saida[MAX2];
} TipoMatriz;
```

Onde:

- `estado` é o estado da transição;
- `saida` é a saida gerada pela transição.

Essa representação é acompanhada por uma lista de estados e uma lista de entradas, conforme estruturas abaixo:

```
struct _noent {
   char elem[MAX2];
   struct _noent *prox;
};
typedef struct _noent noEntrada;
```

Onde:

- `elem` é a entrada,
- `prox` é o ponteiro para o próximo elemento da lista de entradas.

```
struct _noest {
   char elem[MAX];
   char HSI[MAX][MAX3];
   char transition[MAX][MAX3];
   char preamble[MAX];
   struct _noest *prox;
};
typedef struct _noest noEstado;
```

Onde:

- `elem` é o estado da MEF;
- `HSI` é onde será armazenado conjunto HSI daquele estado;
- `transition` é o Transition Cover do estado;
- `preamble` é o Preâmbulo do estado;
- `prox` é o ponteiro para o próximo elemento da lista de estados.

A leitura da MEF se dá pelo seguinte comando:

```
EstadoOrigem - Entrada/Saída -> EstadoDestino
```

A cada execução do comando acima, são inseridos todos os componentes no comando numa lista, cuja estrutura segue abaixo:

```
struct _noEntradaDados {
   char estOrigem[MAX];
   char estDestino[MAX];
   char entrada[MAX];
   char saida[MAX];
   struct _noEntradaDados *prox;
};
typedef struct _noEntradaDados EntradaDados;
```

Onde:

- `estOrigem` é o estado de origem da transição;
- `estDestino` é o estado destino da transição;
- `entrada` é a entrada que fará a transição;
- `saida` é a saida produzida pela transição;
- `prox` é o ponteiro para o próximo elemento da lista de entrada de dados.

Quando se termina a leitura da MEF, os estados e as entradas da lista acima são contados e inseridos em suas respectivas listas. Após esse processo, cria-se a matriz de transição representando a MEF inserida.

Temos na Figura 5 a estrutura para a MEF da Figura 1

Figura 5: Estrutura de representação da MEF da Figura 1.

4.2 Preâmbulo e Transition Cover

Como já foi dito, o *preâmbulo* é a sequência de entradas que leva a MEF do estado inicial para determinado estado e o *transition cover* é o *preâmbulo* concatenado com as entradas aceitas pelo estado atual.

A estrutura de dados utilizada para se obter o *preâmbulo* é uma árvore, conforme estrutura abaixo:

```
typedef struct _no_p {
   char estado[MAX];
   char entrada[MAX];
```

```
    struct _no_p *pai;
    struct _no_p *entradas;
    struct _no_p *lista;
    int valido;
}no_p;
```

Onde:

- `estado` é determinado estado da MEF;
- `entrada` é determinada entrada aceita pela MEF;
- `pai` é o ponteiro que aponta para o pai do nó atual;
- `entradas` é o ponteiro que aponta para o vetor filho;
- `lista` é o ponteiro que faz a ligação entre os vetores filhos;
- `valido` indica se o estado ainda é válido para a criação da árvore.

A raiz é o estado inicial da MEF e o número de filhos dessa árvore é igual ao número de entradas N aceitas pela MEF.

A estrutura é dada por um vetor de tamanho N, onde cada campo é um nó da árvore do tipo `no_p`. Cada nó da árvore terá o ponteiro `entrada` que aponta para o vetor que seu filho faz parte, o ponteiro `lista` que aponta para o próximo vetor do mesmo nível da árvore e o ponteiro `pai`. Essa estrutura pode ser melhor compreendida na Figura 6, que representa a árvore da Figura 2.

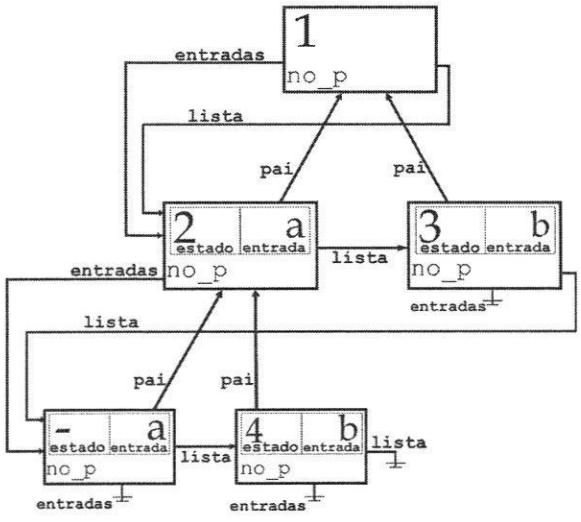

Figura 6: Estrutura da árvore de obtenção do Preâmbulo da MEF da Figura 1.

Depois de construída a árvore, o preâmbulo é obtido através dos nós folhas que, pelo ponteiro `pai`, se faz o caminho até a raiz, invertendo a seqüência encontrada.

Ao ser executado o *preâmbulo* na MEF a ser testada, é verificado o estado em que a execução parou, então é concatenado com esse *preâmbulo* todas as entradas aceitas pelo estado atual, dessa forma obtém-se o *Transition Cover*.

Após serem calculados, esses dois conjunto são inseridos, em seus campos correspondentes, na lista de estados onde serão utilizados posteriormente.

4.3 Conjunto HSI

Para se obter o conjunto HSI, utilizamos três estruturas, as quais seguem abaixo:

```
typedef struct _pontHSI {
   char entrada[MAX];
   struct _noHSI *pont;
}pontHSI;
```

Essa estrutura é cada nó de uma lista encadeada onde são armazenadas todas as entradas aceitas pela MEF, onde:

- `entrada` armazena as entradas aceitas pela MEF;
- `pont` é ponteiro para o próximo nó da lista.

```
typedef struct _seqHSI {
   char entrada[MAX];
   struct _seqHSI *prox;
   struct _seqHSI *phsi;
}seqHSI;
```

Essa estrutura é cada nó da lista onde são armazenados os conjunto HSI de cada par de estados. É a partir dos dados dessa lista que chegamos ao conjunto HSI final, onde:

- `entrada` é a entrada que faz parte de uma das seqüências do conjunto HSI;
- `prox` é o ponteiro para o próximo nó da lista;
- `phsi` é o ponteiro para outro nó dessa própria lista, que fará a concatenação de entradas entre os pares de estado.

```
typedef struct _noHSI {
   char est1[MAX];
   char est2[MAX];
   char flag;
   struct _pontHSI *Phsi;
   struct _seqHSI *Shsi;
}noHSI;
```

Essa é a estrutura principal para a obtenção do conjunto HSI, onde:

- `est1` é um estado da MEF;
- `est2` é um estado da MEF;
- `flag` indica se o par de estados é *fail*;
- `Phsi` é o ponteiro para a lista `pontHSI`;
- `Shsi` é o ponteiro para a lista `seqHSI`.

Os campos `est1` e `est2` formam o par de estados distintos da MEF, onde para cada entrada da lista `pontHSI` é verificado se esta é aceita por ambos os estados. Se for a lista `seqHSI` recebe essa entrada.

4.4 Algoritmo para obtenção do preâmbulo

O Algoritmo 1 monta a árvore utilizada para obter as seqüências do preâmbulo. Nesse algoritmo, são utilizadas algumas funções elementares de estruturas de dados, as quais são:

- *criar_raiz* indica a criação da raíz do preâmbulo;
- *insere_fila* indica a inserção na estrutura fila;
- *remove_fila* indica a remoção na estrutura fila;
- *adiciona_no* indica a inserção de determinado nó na árvore do preâmbulo;

Depois de construída a estrutura do preâmbulo, o conjunto do preâmbulo é encontrado percorrendo a árvore do estado inicial até os nós folhas da árvore.

Algorithm 1 Preâmbulo

```
1: procedure EstruturaPreambulo
2:     covered ← {S_0}
3:     root ← criar_raiz(S_0)
4:     insere_fila(root)
5:     while covered ≠ S do
6:         n ← remove_fila().
7:         for all x ∈ X do
8:             s' ← T(n.estado, x)
9:             if s' ∉ covered then
10:                covered ← covered ∪ {s'}
11:                adiciona_no(n, x, s')
12:                insere_fila(s')
13:            end if
14:        end for
15:    end while
16: end procedure
```

4.5 Algoritmo para obtenção conjunto HSI

O Algoritmo 2 monta a estrutura para se obter o conjunto HSI, e o Algoritmo 3 encontra o conjunto HSI de cada estado da MEF. Observe que $cria\_L_0$ indica a criação da lista L_0 do passo 1 do algoritmo de obtenção do conjunto HSI.

Algorithm 2 HSI

```
1: procedure MontaEstruturaHSI
2:     cria_L_0
3:     for N ∈ L_0 do
4:         for x ∈ X do
5:             if (((N.s1, x) ⊆ D) ∧ ((N.s2, x) ⊆ D)) then
6:                 Y_1 ← O(N.s1, x)
7:                 S_1d ← T(N.s1, x)
8:                 Y_2 ← O(N.s2, x)
9:                 S_2d ← T(N.s2, x)
10:                seja N_2 ∈ L_0 tal que N_2.s_1 = S_1d e N_2.s_2 = S_2d
11:                N.ponteiro[x] ← N_2
12:                N.hsi ← x
13:                if Y_1 ≠ Y_2 then
14:                    N.fail ← true
15:                end if
16:            else
17:                N.ponteiro.[x] ← null
18:            end if
19:        end for
20:    end for
21: end procedure
```

Algorithm 3 HSI

```
1: procedure ENCONTRAHSI
2:     melhorou ← true
3:     while melhorou do
4:         for N ∈ L_0 do
5:             if not N.fail then
6:                 for x ∈ X do
7:                     N_2 ← N.ponteiro[x]
8:                     if (N_2 ≠ null) ∧ (N_2.fail) then
9:                         N.hsi ← x • N_2.hsi
10:                        N.fail ← true
11:                        melhorou ← true
12:                    end if
13:                end for
14:            end if
15:        end for
16:    end while
17:
18:    for s ∈ S do
19:        (HSI_s ← ∅)
20:    end for
21:    for n ∈ L_0 do
22:        if N.fail then
23:            HSI_{n.s1} ← HSI_{n.s1 ∪ N.hsi}
24:            HSI_{n.s2} ← HSI_{n.s2 ∪ N.hsi}
25:        end if
26:    end for
27: end procedure
```

4.6 Algoritmo para obtenção do conjunto final

O Algoritmo 4 é utilizada para a obtenção do conjunto de casos de teste final.

5 Ferramenta Plavis

O projeto Plavis objetiva a criação de uma plataforma para integrar um conjunto de ferramentas que apoia as atividades de VV&T para software de sistemas espacias. O estudo envolve o desenvolvimento de métodos e técnicas para a geração, seleção, execução e análise automatizada de testes. Seu objetivo é trabalhar em em conjunto com institutos que possuem o domínio da tecnologia em pesquisas, de forma a obter a melhor prática para o apoio as atividades de VV&T, estabelecendo uma plataforma aberta quer sirva para ensino, pesquisa e também para o uso dos desenvolvedores no Instituto Nacional de Pesquisas Espaciais (INPE) sendo que, a médio prazo, empresas e instituições acadêmicas

Algorithm 4 Conjunto Final

```
1: procedure EncontroConjuntoFinal
2:     HSI ← {}
3:     for P ∈ preambulo do
4:         for x ∈ X do
5:             S_d ← T(S_0, P)
6:             if (S_d, x) ∈ D then
7:                 C ← P • x
8:                 S_f ← T(S_d, x)
9:                 seq ← C • HSI_{S_f}
10:                HSI ← HSI ∪ seq
11:            end if
12:        end for
13:    end for
14: end procedure
```

utilizem essa ferramenta. A arquitetura da Plavis consiste de ferramentas e técnicas de VV&T, que são utilizadas tanto por indústrias quanto por universidades para a realização de atividades VV&T.

5.1 Plavis/FSM

Atualmente, existe uma versão específica para MEFs, denominada PLAVIS/FSM, que integra os módulos Proteum/FSM [7], Condado [13], MGASet [2] e UIO [15, 1]. Essas ferramentas de suporte a VV&T foram desenvolvidas independentemente por diferentes pesquisadores em diferentes períodos. Com a integração desses módulos e outros que podem ser integrados, espera-se realizar experimentos e comparar as habilidades de cada módulo em revelar falhas nos sistemas testados.

5.1.1 Arquitetura

A arquitetura do Plavis/FSM, conforme ilustrada na Figura 7, foi projetada e desenvolvida como uma aplicação Web[1], possibilitando o uso remoto. O núcleo da plataforma é o módulo *Test Case Manager* (TCM), que engloba as principais funcionalidades da ferramenta coordenando a comunicação com os outros módulos, além de processar as entradas do usuário e retornar as saídas.

Sempre que é necessário interagir com uma das ferramentas integradas, o TCM delega a tarefa a qualquer um dos dois adaptadores. Um adaptador interage com as ferramentas

[1] A ferramenta pode ser acessada através do endereço: `http://143.107.183.158/~adenilso/plavisFSM/main.php`

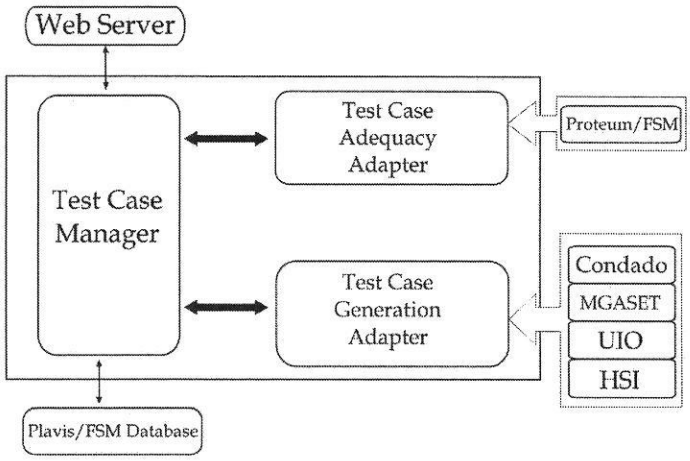

Figura 7: Arquitetura do Plavis/FSM.

para o *Test Case Generation* (TCGA) e o outro interage com as ferramentas para a análise do *Test Case Adequacy* (TCAA). A fim de fazer os adaptadores mais reusáveis, fornece-se somente uma relação pública com as ferramentas que podem interagir com o TCM, entretanto, no caso onde a ferramenta não pode facilmente ser modificada para usar um ou outro adaptador, um envoltório pode ser fornecido, visando a conversão da entrada e saída da ferramenta ao formato que os adaptadores aceitam.

5.1.2 Uso da Plavis/FSM

Primeiramente cria-se um projeto em que a MEF é selecionada, sendo que esta seleção pode ser feita de duas formas: a primeira através do editor de MEFs contido na ferramenta e a outra por inserção da MEF no formato LES (Figura 9), através de um arquivo. Dessa forma a ferramenta gera automaticamente a imagem e o código LES da MEF inserida, conforme exibido na Figura 8. Feito isso, escolhe-se os operadores de mutação, que serão utilizados para gerar os mutantes. Já nessa etapa se tem o número de mutantes gerados de acordo com os operadores de mutação escolhidos.

A próxima etapa é criar uma ou mais sessões dentro do projeto. Depois de criada a sessão, escolhe-se o método de geração para gerar os casos de teste, gerando, conforme Figura 10, e executando na MEF, tendo assim os resultados da execução. Os casos de teste gerados podem ser visualizados, conforme Figura 11.

Abaixo são listados os passos enumerados de forma a facilitar o uso da ferramenta.

1. Autenticação no sistema;
2. Criação do projeto;
3. Seleção dos mutantes;

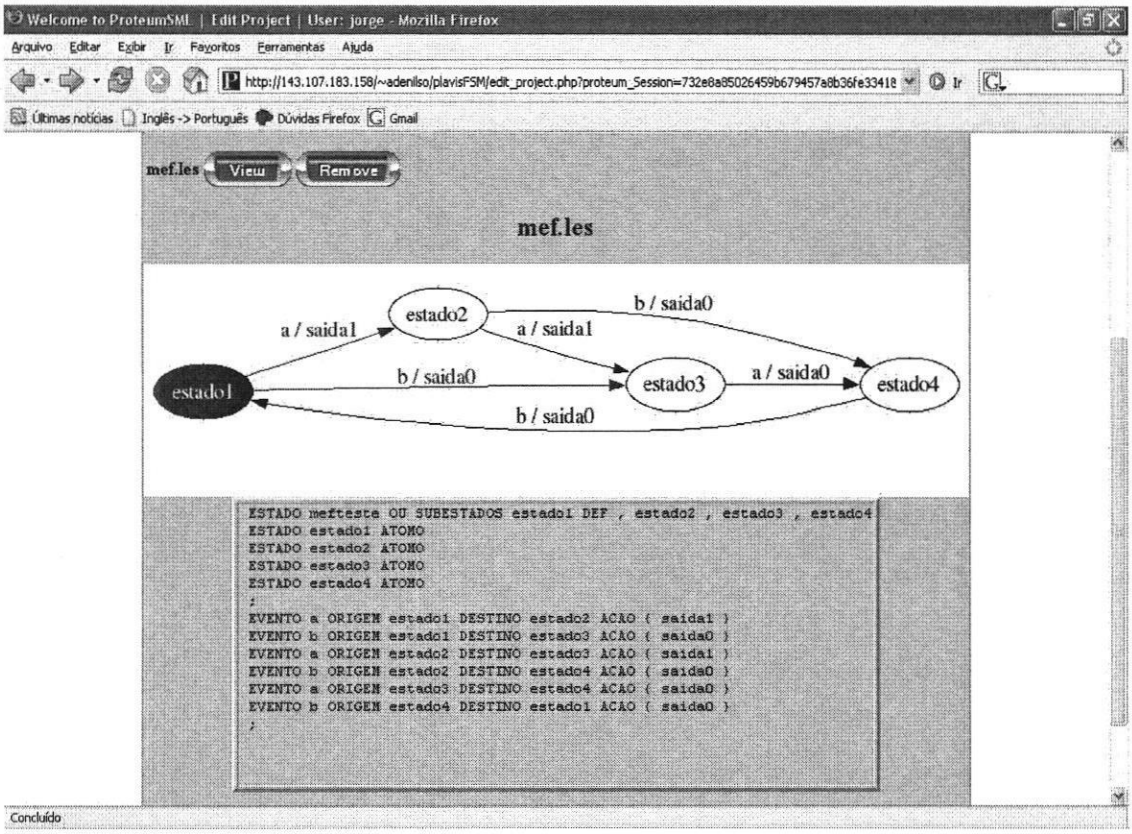

Figura 8: MEF e LES gerados pela ferramenta Plavis/FSM.

4. Inserção da MEF pelo arquivo `.les` ou pelo editor de MEFs da ferramenta;

5. Criação de uma ou mais sessões;

6. Importação dos casos de testes;

7. Análise dos resultados.

5.2 Implantação na ferramenta

Para se fazer a implantação do módulo HSI de geração de casos de teste na Plavis/FSM foi criado o arquivo `class.hsimethod.inc` que se trata da classe do módulo HSI de geração de casos de teste, que foi chamado no arquivo `setup.inc` que por sua vez é chamado em `view_session.php`, onde foi inserida as linhas de código que importavam os casos de teste e a parte que adicionava o botão do método HSI na interface dos *Test Cases*, conforme Figura 12. Além disso o arquivo `FSMtoHSI.pl` foi criado, pois este converte a mef para o formato les.

```
estado mefteste ou
subestados estado1 def, estado2, estado3, estado4
estado estado1 atomo
estado estado2 atomo
estado estado3 atomo
estado estado4 atomo ;
evento a origem estado1 destino estado2 acao {saida1}
evento b origem estado1 destino estado3 acao {saida0}
evento a origem estado2 destino estado3 acao {saida1}
evento b origem estado2 destino estado4 acao {saida0}
evento a origem estado3 destino estado4 acao {saida0}
evento b origem estado4 destino estado1 acao {saida0} ;
```

Figura 9: Formato LES da MEF da Figura 1

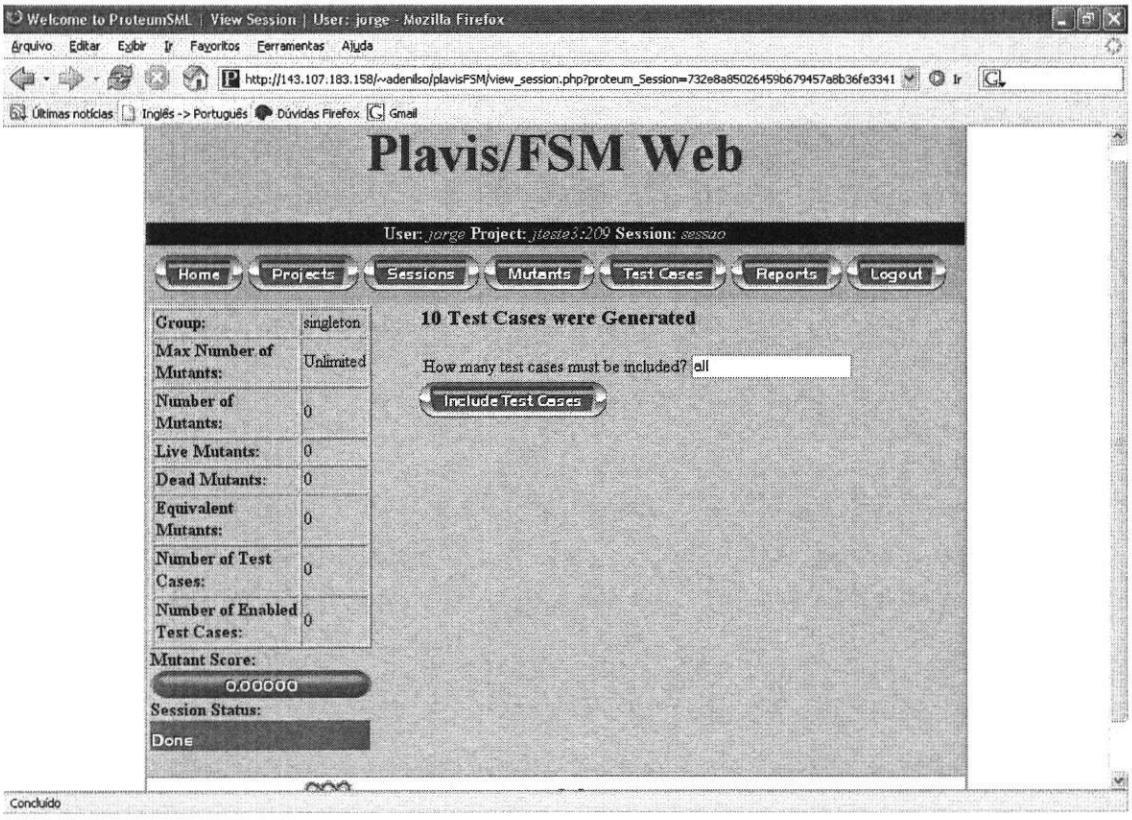

Figura 10: Casos de teste gerados com o método HSI na Plavis/FSM.

6 Dificuldades encontradas

Durante o desenvolvimento do projeto foram encontradas várias dificuldades em várias etapas do desenvolvimento. Pode-se citar como grande dificuldade o entendimento do método HSI, pois eram poucos os artigos que o referenciavam e explicavam seu funcionamento, sendo que isso prejudicou o andamento do projeto.

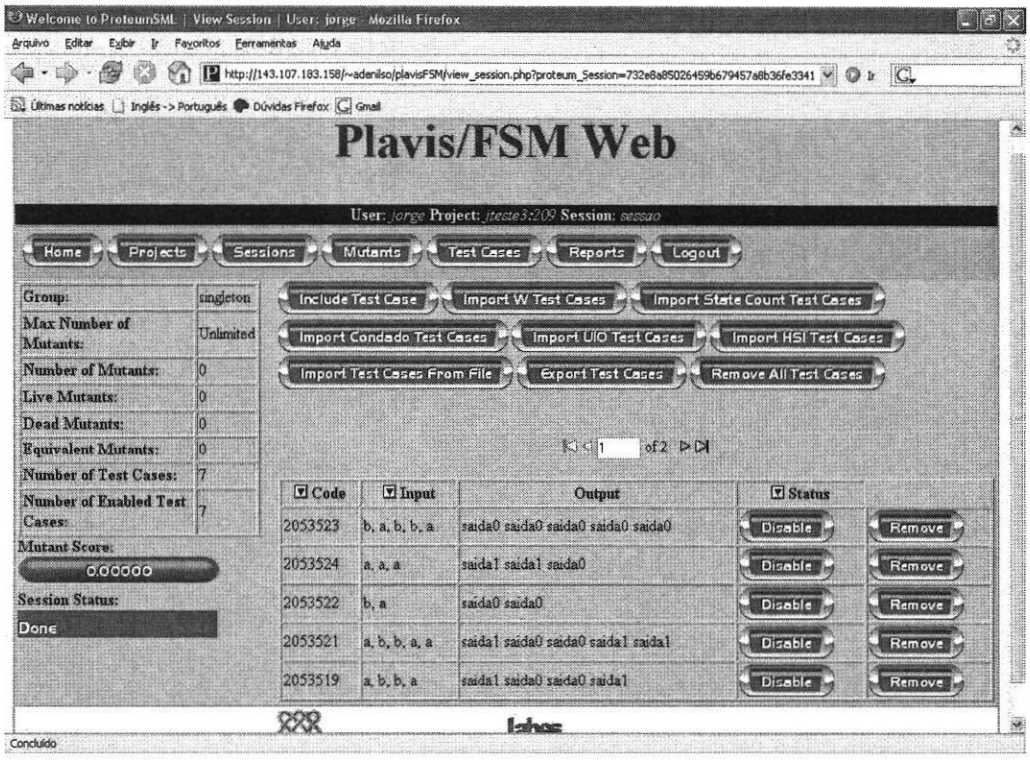

Figura 11: Visualização dos 5 primeiros casos de teste gerados.

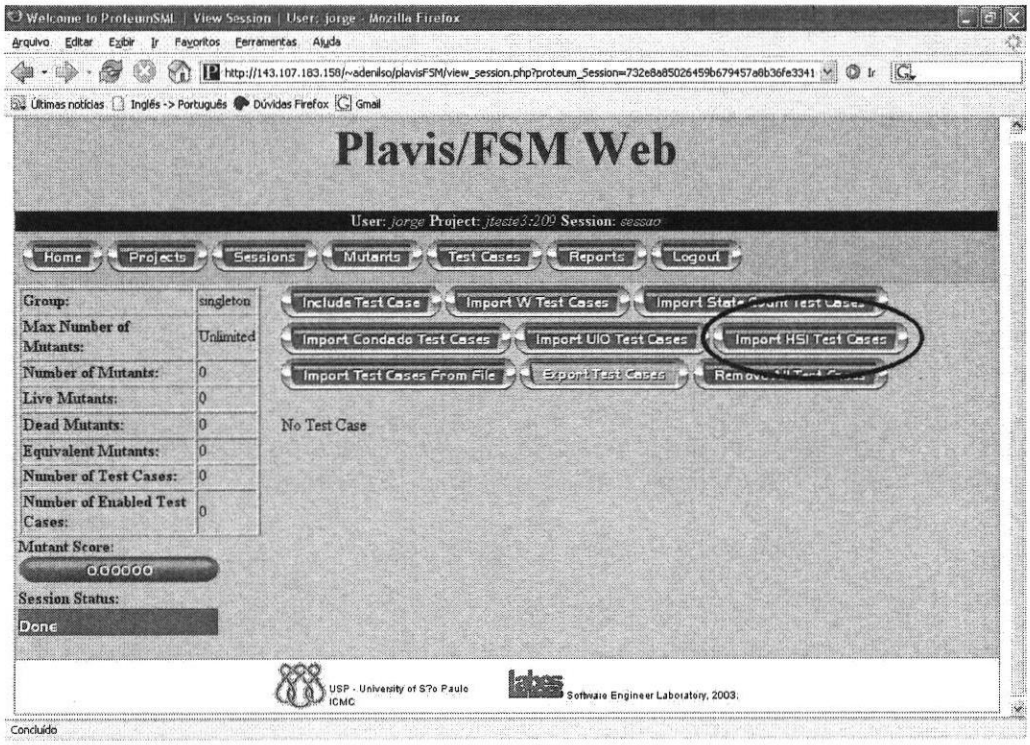

Figura 12: Interface da ferramenta Plavis/FSM na tela da importação de casos de teste.

Outra dificuldade encontrada foi em relação a extensão da ferramenta Plavis/FSM, pois a estrutura do código não favorece a inclusão de novos módulos sem a alteração de páginas já existentes, sendo que um trabalho de reengenharia poderia ser feito para

melhorá-la. Outro problema da Plavis/FSM é a interface pouco intuitiva, o que dificulta o aprendizado e a usabilidade, além de estar muito voltada a Análise de Mutantes [5, 6], o que torna seu uso restrito, já que uma ferramenta como essa poderia trazer várias outras funcionalidades.

7 Conclusão

Para se obter um software de qualidade é imprescindível o uso de ferramentas e técnicas para realizar testes de qualidade. Gerar casos de testes não é uma tarefa fácil, e para isso existem diversas métodos que são capazes de gerar sequências para testes.

Com o uso da ferramenta Plavis/FSM para testes de MEFs, tem-se a necessidade da obtenção de diversos métodos de geração de casos de teste, pois assim pode-se ter um estudo comparativo entre os geradores de sequências, e assim podendo optar pelo melhor método para a MEF testada.

Este trabalho contribuiu com o desenvolvimento de um módulo para a geração de sequências de teste para MEFs parciais com base no método HSI [11, 17]. O módulo foi integrado a Plavis/FSM, permitindo que estudos empíricos sejam conduzidos.

Referências

[1] R. O. Anido and A. Cavalli. Guaranteeing full fault coverage for UIO-based methods. Technical report, June 1995.

[2] M. A. Candolo, A. S. Simão, and J. C. Maldonado. Mgaset - uma ferramenta para apoiar o teste e validação de especificações baseadas em máquinas de estado finito. In *Anais do XV Simpósio Brasileiro de Engenharia de Software*, pages 386–391, October 2001.

[3] T. S. Chow. Testing software design modeled by finite-state-machines. *IEEE Transactions on Software Engineering*, 4(3):178–186, May 1978.

[4] A. M. Davis. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9), September 1988.

[5] R. A. DeMillo. Mutation analysis as a tool for software quality assurance. In *COMPSAC80*, Chicago, IL, October 1980.

[6] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra testing environment. In *Second Workshop on Software Testing, Verification and Analysis*, Baniff, Canadá, 1988.

[7] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Proteum/fsm: A tool to support finite state machine validation based on mutation testing. In *In XIX SCCC - International Conference of the Chilean Computer Science Society*, pages 96–104, 1999.

[8] S. Fujiwara, G. V. Bochman, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[9] A. Gill. *Introduction to the Theory of Finite-State Machines.* McGraw-Hill, New York, 1962.

[10] G. Gönenç. A method for design of fault detection experiments. *IEEE Transactions on Computers*, 19(6):551–558, 1970.

[11] G. Luo, A. Petrenko, and G. v. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. Technical report, Department d'IRO, Université de Montreál, October 1994.

[12] J. C. Maldonado. *Critérios de Teste de Software: Aspectos Teóricos, Empíricos e de Automação.* (Trabalho de Livre Docência), ICMC/USP, São Carlos, SP, 1997.

[13] E. Martins, S. B. Sabião, and A. M. Ambrósio. Condata: a tool for automating specification-based test case generation for communication systems. *Software Quality Journal*, 8(4):303–320, 1999.

[14] A. Petrenko and N. Yevtushenko. Testing from partial deterministic fsm specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, September 2005.

[15] K. K. Sabnani and A. Dahbura. Protocol test generation procedure. *Computer NetWorks and ISDN Systems*, 15(4):285–297, 1988.

[16] A. S. Simão, A. M. Ambrósio, S. C. P. F. Fabbri, A. S. M. S. Amaral, E. Martins, and J. C. Maldonado. Plavis/fsm: An integrated platform for validating fsm based system. In *Latin-American Symposium on Dependable Computing*, 2005. (Submetido).

[17] Q. M. Tan, A. Petrenko, and G. v. Bochmann. A test generation tool for specifications in the form of state machines. Technical report, 1996.

[18] N. Yevtushenko and A. Petrenko. Test derivation method for an arbitrary deterministic automaton. *Automatic Control and Computer Sciences*, 1990.