JPlavisFSM
Manual de Instruções

Arineiza Cristina Pinheiro

Adenilso da Silva Simão

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo

1

Introdução

1.1 JPlavisFSM

A ferramenta JPlavisFSM é uma nova versão da antiga plataforma PLAVIS, resultado do projeto "PLAVIS - Plataforma para Validação e Integração de Software em Sistemas Espaciais", financiado pelo CNPq e desenvolvido entre os anos de 2002 e 2004.

A primeira versão da plataforma integrava as ferramentas: MEGASET, CONDADO e uma implementação do algoritmo UIO, responsáveis pela geração de casos de teste a partir de Máquinas de Estados Finitos (MEFs). Além disso, foi integrado a ferramenta ProteumFSM que possuia a capacidade de criar mutantes de especificação em MEFs, executar uma MEF com relação a um conjunto de casos de teste e avaliar conjuntos de casos de teste, também é incorporada a PLAVIS.

A nova ferramenta, entitulada JPlavisFSM, foi desenvolvida no contexto do trabalho de mestrado da aluna Arineiza Cristina Pinheiro, com apoio financeiro do Capes e da FAPESP (Processo 2010/04001-3). O objetivo da evolução da plataforma consiste em implementar melhorias de usabilidade identificadas durante o processo de maturidade da versão anterior, além de incorporar a possibilidade de integração com novos métodos sem a necessidade de recompilação. A JPlavisFSM foi desenvolvida em Java e disponibiliza os métodos W, UIO, HSI, SPY e Switch-Cover. Foi incorporado também, a geração de mutantes para MEFs e a ferramenta n-Complete (Simão e Petrenko, 2010), que implementa outra maneira de se avaliar os conjuntos de teste manipulados na ferramenta.

Este documento está organizado da seguinte forma: na Seção 2, são apresentados alguns dos principais conceitos relacionados à área de testes baseados em MEFs; na Seção 3, é apresentada a ferramenta JPlavisFSM; na Seção 4, é discutido como utilizar a nova interface de edição de MEFs; e, finalmente, na Seção 5, é exibida a nova interface de criação e edição de seções de teste em detalhes.

Capítulo

2

Conceitos

2.1 Conceitos Básicos

Seguir métodos de desenvolvimento e utilizar ferramentas de engenharia de software podem ajudar no processo de criação de software, porém não excluem a chance de falhas no produto final. Desta forma, atividades de Verificação, Validação e Teste (VV&T) podem ser realizadas a fim de minimizar ao máximo os possíveis problemas no software. O objetivo dessas atividades é garantir tanto a conformidade do produto final com a sua especificação quanto a qualidade do processo envolvido. As atividades são basicamente divididas em: (1) **estáticas** e (2) **dinâmicas**. Atividades estáticas são as que não precisam de artefatos executáveis para serem realizadas, tais como inspeção de software e revisões técnicas. As atividades dinâmicas, por sua vez, necessitam de artefatos que possam ser executados para serem verificados. A atividade de teste se enquadra nessa segunda categoria.

2.2 Teste de Software

O foco da atividade de teste é analisar o comportamento que o programa ou modelo venha a apresentar, confrontando-o com o determinado na sua especificação. Assim, um teste bem elaborado é aquele que encontra defeitos, uma vez que o fato de não encontrar problemas não garante que o software está correto. Em outras palavras, um bom **conjunto**

de teste é aquele que leva o programa a falhar. Neste contexto, pode-se listar alguns dos principais termos usados em teste de software, como:

- **Domínio de entrada** – é o conjunto de todos os possíveis valores que podem ser utilizados para executar o programa P ou sistema em teste (SUT – *system under testing*).
- **Dado de teste** – um elemento do domínio de entrada, ou seja, um valor de entrada do sistema.
- **Caso de teste** – é um par entrada/saída esperada de P. Um dado de teste juntamente com o resultado esperado do programa para aquele dado de teste.
- **Conjunto de teste** – também chamado de conjunto de casos de teste, é o cojunto de todos os casos de teste usados durante a atividade de teste em questão.
- **Oráculo** – é o mecanismo de decisão utilizado para decidir se a saída obtida para um determinado teste corresponde à saída esperada. Pode ser uma ferramenta automatizada ou o próprio testador.

Na literatura é possível encontrar definições distintas para: **defeito**, **engano**, **erro** e **falha**. Segundo Delamaro et al. (2007b), um **defeito** (*fault*) é definido como sendo um passo, processo ou definição de dados incorretos. Um **engano** (*mistake*) é a ação humana que resulta em um defeito. Um **erro** (*error*) decorre da execução de um defeito gerando um estado inconsistente ou inesperado no programa, como valores incorretos de variáveis, memória ou do apontador de instruções. Por fim, pode ocorrer uma **falha** (*failure*) no sistema em decorrência de um erro, que torna visível o problema para o observador por meio de um resultado inesperado do sistema.

O objetivo das técnicas de teste é sistematizar o processo de definição de conjunto de teste, de forma a verificar a adequação do software à sua especificação, garantindo maior qualidade e segurança ao produto final. O teste baseado em máquinas de estados é a técnica de interesse deste trabalho e é abordada em detalhes a seguir.

2.3 Máquinas de Estados Finitos

Uma Máquina de Estados Finitos (MEF, do inglês *finite state machine*) é uma representação de uma máquina composta por estados e eventos, que correspondem a transições entre os estados. Uma transição é caracterizada por dois eventos: um de entrada e um de saída. A máquina pode estar em apenas um estado por vez. Ao ocorrer um evento de

entrada, a máquina pode responder com um evento de saída e uma transição para outro estado (podendo ser para o mesmo estado).

Existem dois tipos de MEFs: (1) Mealy e (2) Moore. Na máquina de Mealy, os eventos de saída estão associados às transições, ou seja, ao ocorrer um evento de entrada, o evento de saída ocorre durante a mesma transição. Na máquina de Moore, os eventos de saída estão associados aos estados; sendo assim, o evento de saída ocorre ao final da transição, no seu estado destino. As máquinas diferem na forma de representação do evento de saída, mas para cada máquina de Mealy existe uma máquina de Moore correspondente. Para métodos de geração de sequências de teste, a máquina mais amplamente utilizada é a de Mealy, a qual corresponde ao foco deste trabalho.

A representação de uma MEF pode ser feita por um diagrama de estados, em que círculos representam os estados e arcos direcionados, as transições. Essa representação se assemelha a de um autômato finito com saídas. Outra representação possível é no formato de tabela de transição, em que os estados são representados por linhas e os estados por colunas. Na Figura 2.1, é apresentada uma MEF de quatro estados e oito transições representada por um diagrama de estados e, na Tabela 2.1, é representado a mesma MEF em forma de tabela de transições.

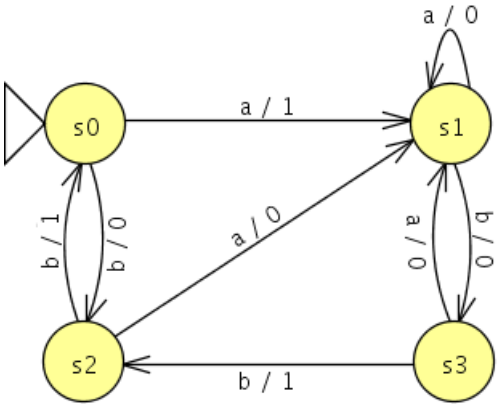

Figura 2.1: Exemplo de MEF.

Tabela 2.1: Tabela de transição da MEF da Figura 2.1.

Entrada \ Saída	a	b	a	b
s0	1	0	s1	s2
s1	0	0	s1	s3
s2	0	1	s1	s0
s3	0	1	s1	s2

Formalmente, pode-se definir uma MEF como uma tupla $M = (S, s_0, X, Y, D, \lambda, \delta)$, sendo:

- S: conjunto finito não-vazio de estados;
- s_0: estado inicial, tal que $s_0 \in S$;
- X: conjunto finito não vazio de símbolos de entrada;
- Y: conjunto finito não vazio de símbolo de saída;
- D: domínio da especificação, tal que $D \subseteq (S \times X)$;
- λ: função de saída, tal que $\lambda : D \rightarrow Z$;
- δ: função de transição, tal que $\delta : D \rightarrow S$.

Considerando a MEF da Figura 2.1, tem-se que para o estado s_0: $\lambda(s_0, a) = 1$ e $\lambda(s_0, b) = 0$ (funções de saída) e $\delta(s_0, a) = s_1$ e $\delta(s_0, b) = s_2$ (funções de transição). Pode-se estender essas notações para sequências de entrada, considerando uma sequência de entrada $\alpha.a$ aplicada a partir do estado s, então $\delta(s,\alpha.a) = \delta(\delta(s,\alpha),a)$ e $\lambda(s,\alpha.a) = \lambda(s,\alpha).\lambda(\delta(s,\alpha),a)$. Por exemplo, para o estado s_0, dada a sequência de entrada $abba$, temos $\lambda(s_0, abba) = 1010$ e $\delta(s_0, a) = s_1$, que define uma sequência de entrada que antige o estado s_1 a partir do estado inicial, gerando a sequência de saída 1010. Outra notação empregada para representar as funções de saída e de transição é $s_i - x/y \rightarrow s_j$, que representa uma transição com entrada x e saída y do estado origem s_i ao estado destino s_j, sendo s_i, $s_j \in S$, $x \in X$ e $y \in Y$. Para representar uma sequência finita de símbolos de entrada, pode-se utilizar a mesma notação sem a necessidade de representar a sequência de símbolos de saída obtidos, por exemplo, temos que $s_0 - abba \rightarrow s_1$.

A notação $\alpha\beta$ representa a concatenação de duas sequências de entrada, sendo α, $\beta \in X^*$, em que X^* representa todas as sequências do domínio de entrada. A notação AB representa a concatenação de conjuntos, tal que $AB = \{\alpha\beta \mid \alpha \in A, \beta \in B\}$.

Dadas duas sequências α e β, α é dita *prefixo* de β, denotado $\alpha \leq \beta$, se $\beta = \alpha\omega$, para algum ω. A sequência α é um *prefixo próprio* de β, denotado $\alpha < \beta$, se $\beta = \alpha\omega$, para algum $\omega \neq \varepsilon$, sendo que ε representa a *sequência vazia*. Uma *sequência de entrada* é chamada de um *caso de teste*, ou apenas *teste*.

Um conjunto de casos de teste T é um conjunto finito de sequências ou testes, tal que não existam duas sequências α, $\beta \in T$ com $\alpha < \beta$. O conjunto de todas as sequências de entrada possíveis a partir do estado s_i da MEF M é denotado $\Omega_M(s_i)$. Caso deseje denotar o conjunto de sequências para s_0, pode-se omitir o estado, correspondendo apenas a Ω_M.Para um conjunto de teste T, a notação $pref(T)$ corresponde ao conjunto de todos os prefixos dos casos de testes contidos em T, isto é, $pref(T) = \{\alpha|\beta \in T, \alpha \leq \beta\}$.

Para executar cada sequência do conjunto de teste T é necessário que a MEF esteja em seu estado inicial. Sendo assim, uma operação que leve corretamente a MEF de volta para

o seu estado inicial antes que a próxima sequência venha a ser executada deve existir, o chamado *reset confiável*. Como é uma operação essencial para a aplicação das sequências de teste, o seu custo deve ser considerado, representando-o por meio do símbolo r no início de cada sequência. O *custo da execução* de um caso de teste é medido pelo *tamanho da sequência* somado a operação de *reset*, assim, temos $\omega(\alpha) = |\alpha|+1$. Por exemplo, se um conjunto de teste é dado por $T = \{aa,bbb,bab\}$, temos o conjunto final sendo $T' = \{raa,rbbb,rbab\}$ de tamanho 11. O custo de execução do conjunto de testes T é denotado $\omega(T)$, dado pela soma do tamanho de todas as sequências contidas em T.

Outras notações relativas ao contexto de MEF também são utilizadas. Dois estados, s_j de M e t_i de I, sendo a MEF $I = (T, t_0, X, Y, C, \Lambda, \Delta)$, são ditos distinguíveis se existe uma seqüência de entrada $\gamma \in \Omega_M(s_j) \cap \Omega_I(t_i)$, chamada de sequência de separação, tal que $\lambda(s_j,\gamma) \neq \Lambda(t_i,\gamma)$. Duas MEFs I e M são ditas distinguíveis (ou distintas) se todo par de estados (s_i,s_m), sendo $s_i \in S_I$ e $s_m \in S_M$, é distinguível.

Analogamente, considerando dois estados, s_i e s_j de M, diz-se que são equivalentes se para todo $\alpha \in \Omega_M(s_i) \cap \Omega_M(s_j)$ se $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$. Também existe equivalência entre máquinas. M é equivalente a outra MEF I caso seus estados iniciais sejam equivalentes. Estendendo esse conceito para MEFs parcais, temos o conceito de *quasi-equivalência*. Dada a MEF parcial I é dita quasi-equivalente a M se $\Omega_M \supset \Omega_I$.

Um domínio de erro $\Im_n(M)$ é o conjunto de todas as implementações possíveis de M definidas sobre o alfabeto de entrada X. Similarmente, $\Im_n(M)$ denota o conjunto de todas MEFs completas definidas sobre o alfabeto de entrada X com no máximo n estados. O conjunto de teste T é dito *n-completo* para a especificação M se para todo $I \in \Im_n(M)$ tal que I é distinguível de M, existe pelo menos uma sequência $\alpha \in T$ que distingue I de M. Se T possui uma única sequência, essa sequência é chamada de sequência de verificação (*checking sequence*).

2.3.1 Propriedades e Características

MEFs possuem algumas propriedades importantes quanto a sua estrutura, como:

- **Completude**: uma MEF é dita **completamente especificada**, ou completa, se ela trata todas as entradas pertencentes ao domínio de entrada (X) em todos os estados (S), tal que $D = (S \times X)$. Caso contrário, a MEF é **parcialmente especificada**, ou parcial;

- **Conectividade**: uma MEF é **fortemente conexa** se para cada par de estados (s_i,s_j) existe um caminho que leva s_i a s_j, ou seja, existe alguma sequência de entrada que executa um caminho de transições com origem em s_i e destino a s_j.

Caso seja possível atingir todos os demais estados a partir do estado inicial, a MEF é dita **inicialmente conexa**;

- **Determinismo**: uma MEF é **determinística** quando há uma única transição para uma dada entrada e qualquer estado do conjunto que permita a transição para um próximo estado. Caso contrário, a MEF é dita **não-determinística**;
- **Equivalência**: um estado s é dito **equivalente** a s', se para todo $\rho \in \Omega_M(s) \cap \Omega_M(s')$, $\lambda(s, \rho) = \lambda(s', \rho)$. Isto é, dois estados são equivalentes se não existe nenhuma sequência de entrada que, ao ser executada a partir dos respectivos estados, gere uma sequência de saída diferente. Utiliza-se a notação $s \approx s'$ para estados equivalentes; e
- **Minimalidade**: uma MEF é **reduzida** (ou minimal) se não existem dois estados equivalentes. Caso contrário, a MEF é dita **não-reduzida**.

Na Figura 2.1, pode-se observar um exemplo de MEF completa, reduzida, fortemente conexa e determinística. Na Figura 2.2, observa-se um exemplo de MEF: parcial, dado que o estado s_5 não trata o evento de entrada b; (2) não-reduzida, pois os estados s_1 e s_4 são equivalentes; (3) não-determinística, uma vez que o estado s_5 possui duas transições possíveis para o evento de entrada a; e (4) desconexa, pois não é possível alcançar o estado s_5 a partir de nenhum estado.

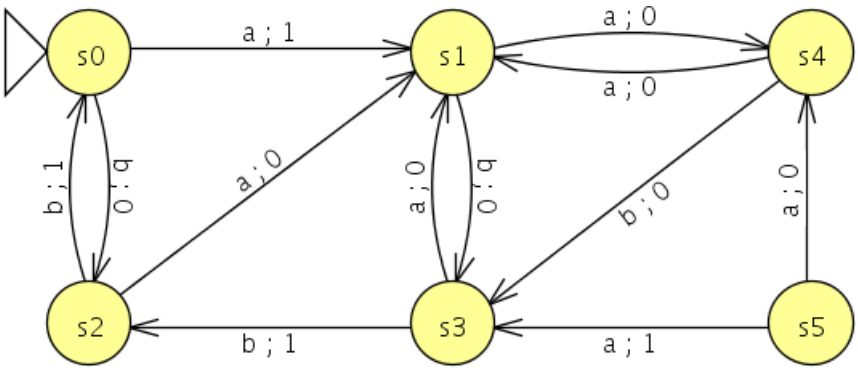

Figura 2.2: MEF parcial, não-reduzida, não-determinística, não-inicialmente-conexa.

2.3.2 Testes Baseados em Máquinas de Estados

No contexto de teste de software, o teste baseado em máquinas de estados finitos tem o objetivo de confrontar uma MEF M reduzida com n estados com a MEF I, que corresponde a MEF implementada. Para tanto, a MEF I também deve ser uma MEF reduzida. A redução é uma característica importante uma vez que garante a inexistência de uma

outra MEF equivalente, com m estados, tal que $m \leq n$. De acordo com Chow (1978), uma MEF minimal M, que representa a versão correta da MEF especificada, se comparada com a MEF I, podem exibir os seguintes erros:

- **Erros de saída**: I e M diferem na saída obtida para um dado teste. Para tornar I equivalente a M, deve-se modificar a função de saída de I.
- **Erros de transferência**: quando I não é equivalente a M, e I. Pode-se tornar equivalente a M ao alterar a função de transição de I.
- **Erro de transição**: termo geral para erro de saída e/ou de transferência.
- **Erro de estado inicial**: I e M não são equivalente. Porém basta alterar o estado inicial de I para solucionar o erro.
- **Estados extras**: quando I possui estados a mais que que M. Para tornar I equivalente a M, basta modificá-la diminuindo o número de estados.
- **Estados ausentes**: erro oposto ao de estados extras, em que I não é equivalente a M por possuir um número de estados inferior. Para torná-la equivalente a M, basta alterar I acrescentando estados.

Na Figura 2.3, são exemplificados erros de operação e transferência, respectivamente, em relação a MEF apresentada na Figura 2.1, que possui a transição $s_2 - b/1 \rightarrow s_0$.

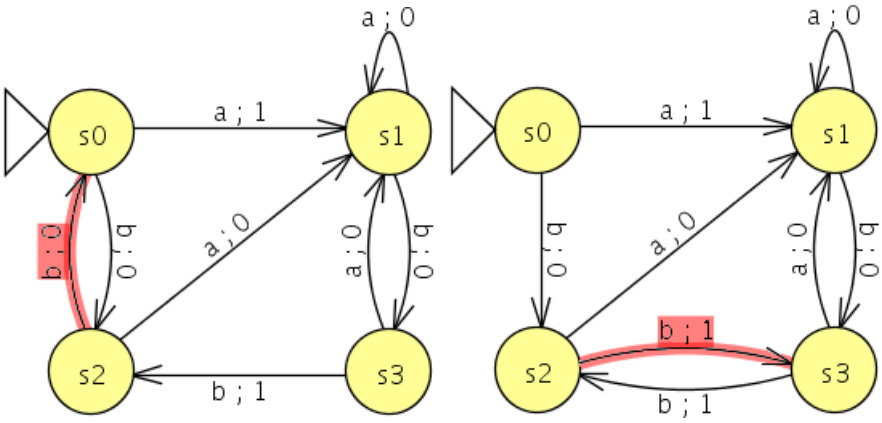

Figura 2.3: (a)Erro de Operação; (b)Erro de transferência.

Para se identificar os erros descritos acima, uma opção seria a realização de teste exaustivo. O conjunto X^*, que representa todas as sequências de entradas possíveis, seria aplicado com o objetivo de verificar se as saídas de I equivalem as saídas de M. Porém, é inviável realizar o teste exaustivo, uma vez que o conjunto X^* é infinito.

Diversos métodos de geração de casos de testes têm sido propostos para a técnica de teste baseado em MEFs, com o objetivo de encontrar um conjunto de sequências que garanta a equivalência das MEFs. O método ideal seria aquele que gera o menor conjunto de teste e que seja suficiente para revelar todos os possíveis erros de uma implementação. A Seção 2.4 apresenta alguns dos métodos de geração de casos de testes propostos para MEFs.

2.4 Métodos de Geração de Casos de Testes

Métodos de geração de sequências de teste têm por objetivo verificar se uma implementação está correta com sua especificação, por meio da execução de atividades de teste e validação em sistemas descritos por modelos (Fujiwara et al., 1991). Apesar de todos definirem procedimentos para a geração de testes, a principal diferença que os evidenciam é o custo da geração dessas sequências e a capacidade de detecção de defeitos (efetividade). Desta forma, deve-se levar em conta a relação custo-benefício de cada método. O foco principal consiste em promover a detecção do maior número possível dos defeitos existentes em uma implementação levando em conta o tamanho do conjunto gerado, para que esse fato não inviabilize a sua aplicação prática.

A maioria dos métodos não gera conjuntos unitários de teste, isto é, com uma única sequência. Para executar cada sequência do conjunto é necessário que a MEF esteja em seu estado inicial. Sendo assim, uma operação que leve corretamente a MEF de volta para o seu estado inicial antes que a próxima sequência venha a ser executada deve existir, o chamado ***reset* confiável**. Como é uma operação essencial para a aplicação das sequências de teste, o seu custo deve ser considerado, representando-o por meio do símbolo r no início de cada sequência. Por exemplo, se um conjunto de teste é dado por $T = \{aa,bbb,bab\}$, temos o conjunto final sendo $T' = \{raa,rbbb,rbab\}$ de tamanho 11.

Como os métodos de geração de casos de testes (ou sequências de teste) são fortemente baseados em sequências básicas, primeiramente são apresentadas algumas das sequências importantes para a aplicação dos métodos tratados no decorrer desta seção.

2.4.1 Sequências Básicas

Sequências básicas possuem propriedades interessantes que são exploradas para a geração de conjuntos de teste. Algumas definem um caminho que passe por todos os estados pelo menos uma vez ou que execute todas as possíveis transições da MEF. Outras têm por objetivo garantir que, ao serem executadas na MEF, possuam respostas diferentes para

cada um de seus estados. Existem ainda sequências que levam a MEF para determinados estados.

Mais de uma sequência básica podem ser encontradas em uma MEF e, comumente, é aconselhável selecionar a menor sequência básica. Porém nem sempre a menor sequência encontrada é a responsável pelos melhores resultados. Exemplificando com base na Figura 2.1, as sequências mais citadas na literatura são:

- ***State Cover*** (Q): é um conjunto de n sequências de entrada (incluindo a sequência vazia ε), em que n corresponde ao número de estados da MEF M, que ao serem executadas, a partir do estado inicial, terminam em cada um dos estado de M uma vez. Para a Figura 2.1, um exemplo do conjunto *State Cover* seria: $Q = \{\varepsilon,a,b,ab\}$.

- ***Transition Cover*** (P): é um conjunto de sequências de entrada que exercitam cada uma das transições de M, a partir do estado inicial. O *Transition Cover* pode incluir o *State Cover*, pois para incluir todas as transições é necessário que todos os estados sejam visitados. Para a MEF de exemplo, temos: $P = \{\varepsilon,a,b,ab,bb,ba,aa,aba,abb\}$.

- **Sequência de Sincronização** (SS): é uma sequência de entrada $\alpha \in X^*$ que permite alcançar um dado estado independente de qual estado a MEF esteja inicialmente. A sequência SS pode não existir e não é necessariamente única. Para o exemplo dado, são sequências de sincronização: $SS_0 = \{abbb\}$, $SS_1 = \{a\}$, $SS_2 = \{abb\}$ e $SS_0 = \{ab\}$, respectivamente para os estados de mesmo índice.

- **Sequência de Separação**: são sequências que diferenciam um estado s_i de outro estado s_j. Dado que o conjunto formado por todas as sequências possíveis para um estado s é representado por $\Omega_M(s)$, deve existir uma sequência $\alpha \in \Omega_M(s_i) \cap \Omega_M(s_j)$, tal que $\lambda(s_i,\alpha) \neq \lambda(s_j,\alpha)$. Essas são as sequências básicas nas quais se baseiam os métodos de geração de casos de teste. Existem diversos tipos de sequências de separação, a serem descritas nos tópicos a seguir.

- **Sequência Única de Entrada e Saída** (UIO): é uma sequência de entrada/saída única para cada estado s_i, que pode também não existir como as demais sequências de separação apresentadas. Com a aplicação da sequência UIO é possível distinguir o estado s_i de qualquer outro estado, pois a saída produzida é específica do estado s_i. Uma sequência DS é uma UIO para todos os estados. Porém, como o objetivo é utilizar sequências menores, pode ser mais apropriado gerar UIOs, pois ,em geral, é possível obter sequências mais curtas. Para a MEF exemplo da Figura 2.1, temos: $UIO_0 = \{a\}$, $UIO_1 = \{bbb\}$, $UIO_2 = \{bb\}$ e $UIO_3 = \{bb\}$.

- **Conjunto de Caracterização** (W): como DS e UIO, o W é utilizado para distinguir aos pares os estados de uma MEF. É um conjunto de sequências de entrada tal que, para dois estados distintos quaisquer s_j e s_i, existe $\alpha \in W$ tal que $\lambda_M(s_j,\alpha) \neq \lambda_M(s_i,\alpha)$. A a união de todas as sequências UIO formam um conjunto W, assim como a DS representa um W unitário. Por sua vez, o W pode gerar um conjunto menor que as demais sequências descritas. Para a MEF de exemplo, temos: $W = \{a,bb\}$. Como pode-se observar na Tabela 2.2, a sequência bb apenas distingue os estados s_2 e s_3. Porém, ao aplicar em conjunto a sequência a, os estados s_0 e s_1 passam a ser distinguíveis, ou seja, geram saídas diferentes.

Tabela 2.2: W: Saídas obtidas para a MEF da Figura 2.1.

Estados	a	b	b
s0	1	0	1
s1	0	0	1
s2	0	1	0
s3	0	1	1

- **Conjunto de Identificação** (W_p): é um subconjunto de W definido para cada estado s_i de M que distingue o estado s_i dos demais, sendo $W_i \subseteq \Omega_M(s_i)$ o conjunto de seqüências de entrada definidas para o estado s_i, se para qualquer outro estado s_j existe $\alpha \in W_i \cap \Omega_M(s_j)$ tal que $\lambda_M(s_i, \alpha) \neq \lambda_M(s_j, \alpha)$. Por exemplo, para a MEF da Figura 2.1, a sequência a distingue o estado s_0 de todos os demais, então, $W_0 = \{a\}$. Da mesma forma, a sequência de entrada bb, distingue tanto o estado s_2 quanto s_3, sendo $W_2 = W_3 = \{bb\}$. Porém, o estado s_1 não é distinguível apenas com a sequência a ou com bb, mas é distinguível com o conjunto $W_1 = \{a,bb\}$.

- **Conjunto de Identificadores Harmonizados** (H_i): é um conjunto de identificadores de estados tal que, para dois estados quaisquer s_i, $s_j \in S$, $i \neq j$, existe $\alpha \in H_i$ e $\gamma \in W_j$ que têm um prefixo comum β tal que $\beta \in \Omega_M(s_i) \cap \Omega_M(s_j)$ e $\lambda_M(s_i, \beta) \neq \lambda_M(s_j, \beta)$. Sendo assim, temos para a MEF da Figura 2.1: $H_0 = \{a,b\}$, $H_1 = \{bb\}$, $H_2 = \{a,b\}$ e $H_3 = \{bb\}$.

2.4.2 Método W

Um dos métodos mais difundidos para a geração de sequências de testes, o método *Automata Theoretic* ou, como ficou mais conhecido, método W foi proposto por Chow (1978). O nome atribuído ao método originou-se devido à referência feita por Chow (1978) ao conjunto de caracterização, utilizado pelo método, como conjunto W.

O método W pode ser considerado o método clássico e precursor da área, uma vez que a maioria dos trabalhos seguintes foram baseados no método W. Uma restrição quanto ao

método é em relação à sua aplicabilidade, que exige que a MEFs sejam: determinísticas, completas, inicialmente conexas e minimais.

O método W se baseia nos conjuntos W (conjunto de caracterização) e P (*transition cover*). A partir do conjunto W, é gerado um novo conjunto W' que contém um conjunto de sequências capaz de identificar qual é o estado testado da máquina. Para isso, é necessário estimar o número de estados que a MEF I (implementação) tem a mais do que a MEF M (especificação). Considerando n o número de estado da MEF M, estima-se m como sendo o número de estados para a MEF I. Sendo assim, temos:

$$W' = \bigcup_{i=0}^{m-n}(X^i W),$$

em que $X^0 = \{\varepsilon\}$ e $X^i = X.X^{i-1}$. Para o caso $n = m$, temos que $W = W'$.

Definido o conjunto W', as sequências obtidas são concatenadas com o conjunto P, isto é, $T_W = P.W'$. Logo, ao obter as sequências de teste, estas são executadas uma a uma na máquina, gerando as respectivas saídas que serão analisadas a procura de inconsistências entre as MEFs. Como o W é um método completo (Chow, 1978), caso nenhuma falha ocorra, pode-se afirmar que a MEF I está adequada a especificação.

Para o exemplo da MEF M da Figura 2.1, considerando o caso $n = m$, temos: $W' = W = \{a,bb\}$, $P = \{\varepsilon,a,b,ab,bb,ba,aa,aba,abb\}$ e $T_W = \{aaa,baa,bba,aabb,abaa,abba,babb,bbbb,ababb,abbbb\}$. Porém, como dito anteriormente, o conjunto de teste é formado apenas por sequências que não sejam prefixo de outras. Desta forma, pode-se reduzir o conjunto de teste T_W para: $T_W = \{aaa,baa,bba,aabb,abaa,abba,babb,bbbb,ababb,abbbb\}$ de tamanho 39. Considerando as operações de *reset*, temos: $T_W = \{raaa,rbaa,rbba,raabb,rabaa,rabba,rbabb,rbbbb,rababb,rabbbb\}$ de tamanho final 49.

2.4.3 Método Wp

O método W parcial (Wp), do inglês *partial* W, foi proposto por Fujiwara et al. (1991) como uma melhoria do método W. A partir do conjunto W é criado o **conjunto de identificação** W_i, que extrai um subconjunto do conjunto W capaz de identificar cada estado da MEF, dependendo do estado final s_i que foi alcançado pela sequência. Desta forma, o método Wp se baseia nos conjunto Q, P e W_i, sendo realizado em duas fases:

- **Fase 1**: gerar as sequências formadas por r.Q.W
- **Fase 2**: gerar as sequências formadas por $r.R\otimes W$, em que $R = P\backslash Q$ e $R\otimes W = \bigcup_{p\in R}\{p\}.W_i$ tal que i é definido pelo estado final da execução da sequência p.

Para a utilização do método Wp, a MEF deve apresentar as propriedades: determinística, completa, inicialmente conexa e minimal. Da mesma forma que o método W, o Wp foi provado ser completo e por, na segunda fase, concatenar as sequências com um subdomínio de W, apresenta um conjunto final T menor.

Para a MEF da Figura 2.1, temos: $Q = \{\varepsilon,a,b,ab\}$, $P = \{\varepsilon,a,b,ab,bb,ba,aa,aba,abb\}$, $W_0 = \{a\}$, $W_1 = \{a,bb\}$, $W_2 = \{bb\}$ e $W_3 = \{bb\}$. Assim na fase 1, temos: $T_{F1} = \{aa,ba,bb,aba,abbb\}$. Na fase 2, são gerados os conjuntos: $R = \{bb_0,ba_1,aa_1,aba_1,abb_2\}$, em que os índices de cada sequência indicam o estado final atingido por cada uma delas; e $T_{F2} = \{bba,baa,babb,aaa,aabb,abaa,ababb,abbbb\}$. Ao final, é realizada a união de T_{F1} com T_{F2}, eliminando as sequências que são prefixos e inserindo a operação de reset antes de cada sequência selecionada, temos: $T_{Wp} = \{rbba,rbaa,raaa,rbbb,rbabb,raabb,rabaa,rababb,rabbbb\}$ de tamanho 43.

2.4.4 Método UIO

O método UIO foi originalmente proposto por Sabnani e Dahbura (1988) como um método completo. Porém, Vuong et al. (1989) apresentaram um contra-exemplo que desmostrou que o método UIO não garante a cobertura completa de defeitos para todas as MEFs. O método é baseado em sequências UIO e é realizado em apenas uma fase, correspondente a fase 2 do método W, pertinente a fase de teste das transições.

Da mesma forma que o método W, a aplicabilidade do método UIO está restrita existência das sequências UIO e à MEFs que sejam: determinísticas, inicialmente conexas, completas e minimais. A fase de geração do conjunto de teste é definida por:

- **Fase**: gerar as sequências formadas por $r.P \otimes UIO$, em que $P \otimes UIO = \bigcup_{p \in P}\{p\}.UIO_i$ tal que i é definido pelo estado final da execução da sequência p.

Para exemplificar o método UIO, considerando a MEF da Figura 2.1 temos: $P = \{\varepsilon_0,a_1,b_2,ab_3,bb_0,\ ba_1,aa_1,aba_1,abb_2\}$, em que os índice apresentados nas sequências correspondem ao estado final alcançado por cada uma, $UIO_0 = \{a\}$, $UIO_1 = \{bbb\}$, $UIO_2 = \{bb\}$ e $UIO_3 = \{bb\}$. Após a fase de geração do conjunto de teste, eliminando as sequências que são prefixos e inserindo a operação de reset, obtem-se $T_{UIO} = \{rbbb,rbba,rbabbb,raabbb,rababbb,rabbbb\}$ de tamanho 33.

Outros métodos que utilizam a sequência UIO como sequência de separação foram propostos, mas também não garantem a cobertura completa de defeitos. Apenas no trabalho proposto por Vuong et al. (1989) que houve a preocupação de se estabelecer uma prova formal para uma melhoria do atual método UIO, chamado de Método UIOv, a ser trabalho a seguir.

2.4.5 Método UIOv

Como uma variação do método UIO, o método UIOv (UIO *variation*) foi proposto por Vuong et al. (1989). A solução apresentada inclui uma fase a mais no processo de geração de testes, em que seriam aplicada todas as sequências UIO a todos os estados pelo menos uma vez, ou seja, a sequência UIO_i será aplicada em cada estado s_j atingido a partir do conjunto Q, tal que i=$\{0..n\}$ e j=$\{0..n\}$. Desta forma, o método UIO difere do método UIOv na aplicação da primeira fase do processo:

- **Fase 1**: gerar as sequências formadas por r.Q.UIO
- **Fase 2**: gerar as sequências formadas por $r.P \otimes UIO$, em que $P \otimes UIO = \bigcup_{p \in P}\{p\}.UIO_i$ tal que i é definido pelo estado final da execução da sequência p.

Devido a esse novo processo que foi definido a fim de prover cobertura completa para defeitos em MEFs, o método UIOv pode ser considerado um caso especial do método Wp. Isso se torna verdade quando o conjunto W_i possui somente uma sequência simples (UIO) para cada estado s_i da MEF, tornando o método Wp equivalente ao método UIOv.

Para a MEF da Figura 2.1, temos: $Q = \{\varepsilon,a,b,ab\}$, $P = \{\varepsilon_0,a_1,b_2,ab_3,bb_0,\ ba_1,aa_1,aba_1,\ abb_2\}$, em que os índice apresentados nas sequências correspondem ao estado final alcançado por cada uma, na primeira fase, $UIO_0 = \{a\}$, $UIO_1 = \{bbb\}$, $UIO_2 = \{bb\}$ e $UIO_3 = \{bb\}$. Assim, na primeira fase obtem-se $T_{F1} = \{a,bbb,bb,bb,aa,abbb,abb,abb,ba,bbbb,bbb,bb,aba,abbbb,abbb,abbb\}$ e $T_{F2} = \{bba,babbb,aabbb,ababbb,abbbb\}$. Desta forma, ao final de todo o processo de união e redução temos: $T_{UIOv} = \{rbbbb,rbba,rbabbb,raabbb,rababbb,rabbbb\}$ de tamanho 34.

2.4.6 Método HSI

O método HSI (Petrenko et al., 1993), assim como a maioria dos métodos propostos, foi proposto como uma melhoria para o método W. Além de gerar conjunto de teste completo, o método apresenta um grau de aplicabilidade maior do que os demais métodos, pois pode ser aplicado tanto em MEFs completas quanto em MEFs parciais. Desta forma, o método consegue cobrir um número maior de especificações com um custo menor quando comparado com o método W.

Fazendo um paralelo entre os métodos Wp e HSI, ambos tem sequências de separação por estado que possuem o objetivo de distinguir o estado s_i dos demais. Desta forma, a diferença entre os conjuntos W_i e H_i está na forma como são construídos. Enquanto W_i é obtido a partir do conjunto W, o H_i é construído a partir de sequências de separação

que distinguem cada par de estados da MEF. O método HSI consiste basicamente na aplicação das fases:

- **Fase 1**: gerar as sequências formadas por $r.Q \otimes HSI$, em que $Q \otimes HSI = \bigcup_{q \in Q}\{q\}.H_i$ tal que i é definido pelo estado final da execução da sequência q.
- **Fase 2**: gerar as sequências formadas por $r.P \otimes HSI$, em que $P \otimes HSI = \bigcup_{p \in P}\{p\}.H_i$ tal que i é definido pelo estado final da execução da sequência p.

Com relação à MEF da Figura 2.1, temos: $Q = \{\varepsilon_0,a_1,b_2,ab_3\}$, $P = \{\varepsilon_0,a_1,b_2,ab_3,bb_0, ba_1,aa_1,aba_1, abb_2\}$, $H_0 = \{a,b\}$, $H_1 = \{bb\}$, $H_2 = \{a,b\}$ e $H_3 = \{bb\}$. Desta forma, temos na primeira fase $T_{F1} = \{a,b,abb,ba,bb,abbb\}$ e $T_{F2} = \{a,b,abb,ba,bb,abbb,bba,bbb,babb,aabb, ababb,abba,abbb\}$. Ao final do processo obtem-se o cojunto de teste $T_{HSI} = \{rbba,rbbb,rbabb, raabb,rababb,rabba,rabbb\}$ de tamanho 34.

2.4.7 Método Switch Cover

O método Switch Cover (Pimont e Rault, 1976) usa uma abordagem de busca em profundidade e percorre todas as combinações de transições pelo menos uma vez. O método Switch Cover utiliza o conceito de árvore de transição (*transition tree*), na qual a raiz é o estado inicial, para cada transição é desenhado um galho e para cada próximo estado é adicionado um nó. Em caso de loops, é considerado apenas uma iteração. O nó folha é o estado inicial ou um estado já adicionado na árvore. Esse procedimento é repetido até todas as combinações de transições serem exercitadas.

Esse método é exaustivo, pois ele cobre todos os caminhos atingíveis pelo estado inicial, enquanto os demais métodos se baseiam somente nas transições (conjunto P). O conjunto de teste gerado por esse método cresce exponencialmente com o tamanho da MEF, porém não exige que a MEF seja completamente especificada. O método não requer uma fase para identificação dos estados (sequência de separação) como os demais métodos, como o método W que necessita calcular o cojunto de caracterização para gerar o conjunto de teste.

Com relação à MEF da Figura 2.1, um possível conjunto gerado pelo método Switch Cover é $T_{HSI} = \{raabbbabbbabbbaabbbabbbabb\}$ de tamanho 26.

2.5 Teste de Mutação

O Teste de mutação, ou **análise de mutantes**, baseia-se em duas hipóteses (Delamaro et al., 2007a): (1) do programador competente e (2) do efeito de acoplamento. A hipótese do

programador competente assume que programadores experientes desenvolvem programas corretos ou próximos do correto. Desta forma, o critério de mutação parte do princípio que o programa a ser testado é muito próximo do correto e que pequenos defeitos que não geram erros sintáticos podem ter sido inseridos acidentalmente pelos programadores; porém, esses defeitos alteram a semântica do software (Delamaro et al., 2007a). Quanto à hipótese do efeito de acoplamento, assume-se que erros complexos estão relacionados a erros simples. Uma vez que experimentos empíricos evidenciam a veracidade dessa hipótese (Delamaro et al., 2007a), espera-se que casos de teste que encontrem defeitos simples também sejam capazes de encontrar defeitos complexos.

O critério explora essas duas hipóteses assumindo que, ao criar variações simples do artefato de teste por meio da inserção de pequenos defeitos sintáticos, a execução de testes que são capazes de identificar essas alterações nos novos artefatos gerados também será eficiente para identificar problemas mais graves que podem existir no artefato original.

Segundo Mathur (2008), o critério pode ser utilizado como técnica caixa preta. Neste caso, pode ser realizada mutação de especificação e, no domínio de aplicações *web*, mutação de mensagens entre cliente e servidor. Por outro lado, ao considerar o critério de mutação que requer acesso a todo ou parte do código fonte, este é considerado como uma técnica caixa-branca ou baseada no código. Pode-se aplicar o critério para avaliar e melhorar teste para unidades de programas e também aplicá-lo em teste de integração de um conjunto de componentes. Portanto, análise de mutação é uma técnica interessante para se utilizar nas fases de teste de unidade e integração.

Como a análise de mutantes se baseia fortemente nos enganos comumente cometidos e em operadores de mutação sobre o artefato a ser testado, outra possibilidade é aplicar análise de mutantes em modelos de software. Devido à flexibilidade do critério, pode-se aplicá-lo em qualquer tipo de artefato executável, não apenas em programas (Delamaro et al., 2007a). Dessa forma, basta definir os operadores de mutação pertinentes ao novo domínio de teste e um modo de avaliar se o resultado produzido pelo artefato original é o mesmo de cada um dos seus mutantes. Para o contexto de MEFs foram definidos nove operadores de mutação, sendo (Fabbri et al., 1999):

- **OutAlt**: saída trocada. Altera a saída gerada pelos eventos pelas demais saídas existentes;
- **OutDel**: saída faltando. Este operador remove o símbolo de saída associado a cada evento;
- **StaDel**: estado faltando. Suprime um estado;
- **TraArcDel**: arco faltando. Este operador exclui um arco da MEF;

- **TraDesStaDel**: destino trocado. É alterado o estado destino associado a cada um dos eventos pelos demais estados existentes na MEF;
- **TraEveAlt**: evento trocado. É trocado cada evento da MEF pelos demais eventos existentes;
- **TraEveDel**: evento faltando É excluído um evento por vez que provoque a transição entre dois estados;
- **TraEveIns**: evento extra. É incluído, em cada arco da MEF, cada um dos outros eventos existentes que não provoca a transição entre os estados relacionados ao arco considerado; e
- **TraIniStaDel**: alteração do estado inicial da MEF, de forma que em cada mutante um dos outros estados passa a ser o estado de inicialização.

Eventualmente, algum mutante e o modelo original podem apresentar sempre o mesmo resultado, para qualquer caso de teste elaborado. Cabe ao testador verificar e determinar se há equivalência entre os artefatos. Em caso afirmativo, o mutante é classificado como **equivalente**.

Após a execução dos mutantes e a análise de equivalência, é calculado o **escore de mutação**. O objetivo é estimar a adequação dos casos de teste utilizados em uma escala de 0 a 1, fornecendo uma medida objetiva de quão próximo o conjunto está da adequação do critério. Dado o programa P e o conjunto de teste analisado T, é calculado o escore de mutação $ms(P,T)$ como (Delamaro et al., 2007a):

$$ms(P,T) = \frac{DM(P,T)}{M(P)-EM(P)}$$

onde:
$DM(P,T)$: número de mutantes mortos pelo conjunto de casos de teste T;
$M(P)$: número total de mutantes gerados a partir do programa P; e
$EM(P)$: número de mutantes gerados que são equivalentes a P.

O escore de mutação é obtido a partir da razão entre o número de mutantes mortos pelo conjunto T e o número de mutantes que se pode matar, dado pela diferença entre o número total de mutantes gerados e o número de mutantes classificados como equivalentes. Entretanto, além do problema da equivalência, outro grande problema para a aplicação do critério análise de mutantes está relacionado ao seu alto custo de execução, uma vez que o número de mutantes gerados pode ser muito grande, o que dispende um alto tempo de execução (Delamaro et al., 2007a).

Capítulo

3

A Ferramenta

Esta seção apresenta as funcionalidades básicas disponibilizadas pela ferramenta. Será dedicado posteriormente seções mais detalhadas sobre as principais funcionalidades como Edição de MEFs (Seção 4) e Criação/Edição de Sessões de Teste (Seção 5)

3.1 Menu File

No menu *File* (Figura 3.1), estão disponibilizadas as funcionalidades de gerenciamento dos arquivos de MEF da ferramenta (extensão **.jff**). É possível, por exemplo, abrir uma nova área de edição de MEFs (*New...*); abrir um arquivo de uma MEF salva (*Open...*); ou Salvar uma MEF (*Save* e *Save As...*).

3.2 Menu Test Session

No menu *Test Session* (Figura 3.2), pode-se criar uma nova sessão de teste ou abrir uma sessão existente para consulta/edição. Na Seção 5 será apresentado em detalhes como criar/editar uma sessão de teste.

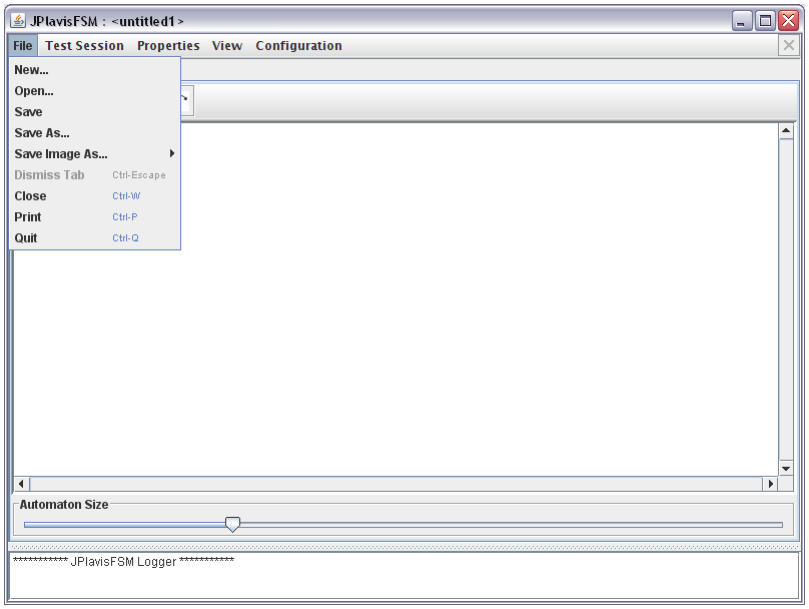

Figura 3.1: JPlavisFSM - Menu *File*.

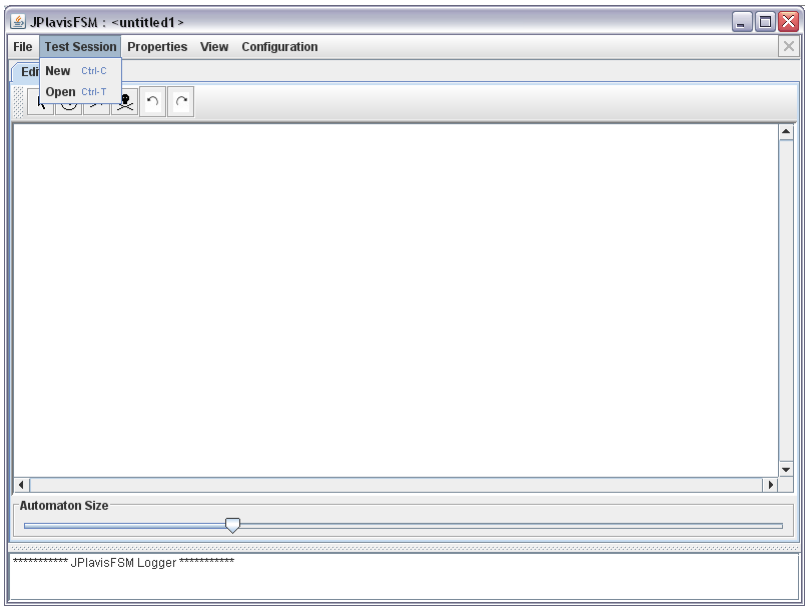

Figura 3.2: JPlavisFSM - Menu *Input*.

3.3 Menu Properties

No menu *Properties* (Figura 3.3), é possível verificar algumas propriedades[1] das MEFs criadas na ferramenta, como: (1) a existência de não-determinismo (*Highlight Nondeterminism*); (2) a existências de transições espontâneas, ou seja, sem a necessidade da ocorrência de um evento de entrada (*Highlight λ-Transitions*); (3) se a MEF é completamente especificada, evidenciando os estados incompletos (*Highlight Partially Specified*

[1]Ver Seção 2.3.1

States); (4) se a MEF é inicialmente conexa, apontando quais os estados não podem ser alcançados a partir do estado inicial (*Highlight Inicially Disconnected States*); e, (5) se a MEF é reduzida, indicando quais estados (aos pares) são equivalentes entre si (*Highlight Equivalent States (unreduced FSM)*).

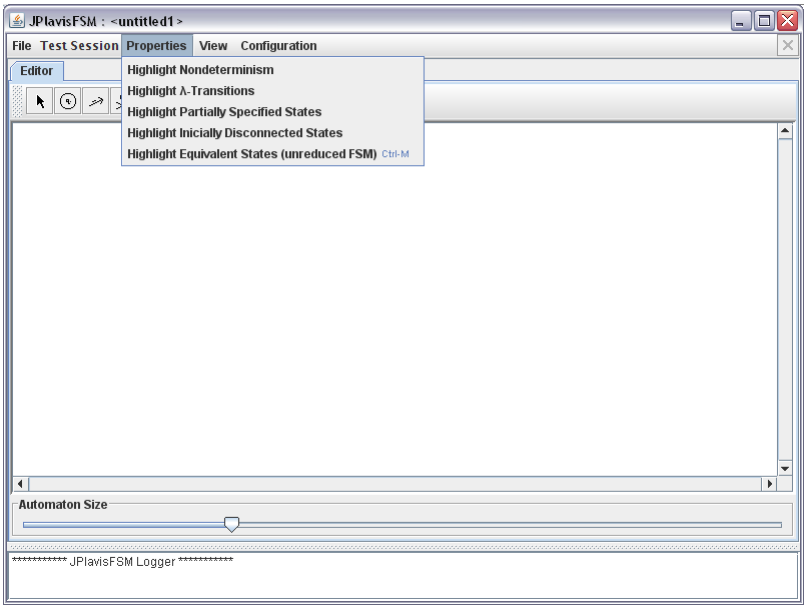

Figura 3.3: JPlavisFSM - Menu *Properties*.

3.4 Menu View

O menu *View* (Figura 3.4) possibilita a alteração do layout da MEF em edição.

3.5 Menu Configuration

Para a geração de sequências de teste, a ferramenta utiliza o conceito de separadores de entrada, uma vez que uma entrada pode ser representada por um único caractere ou por uma palavra. No menu *Configuration* → *Input Separator* (Figura 3.5), é possível escolher qual separador será utilizado durante a edição de seções de teste. Esta opção visa facilitar a inclusão de casos de teste gerados por fontes externas a ferramenta, padronizando o tratamento das sequências.

O usuário pode escolher utilizar o separador: (1) padrão ('vírgula'); (2) ponto; (3) nenhum, considerando entradas como caracteres; ou (4) outro, sendo que o usuário deve informar qual o caractere a ser utilizado (Figura 3.6).

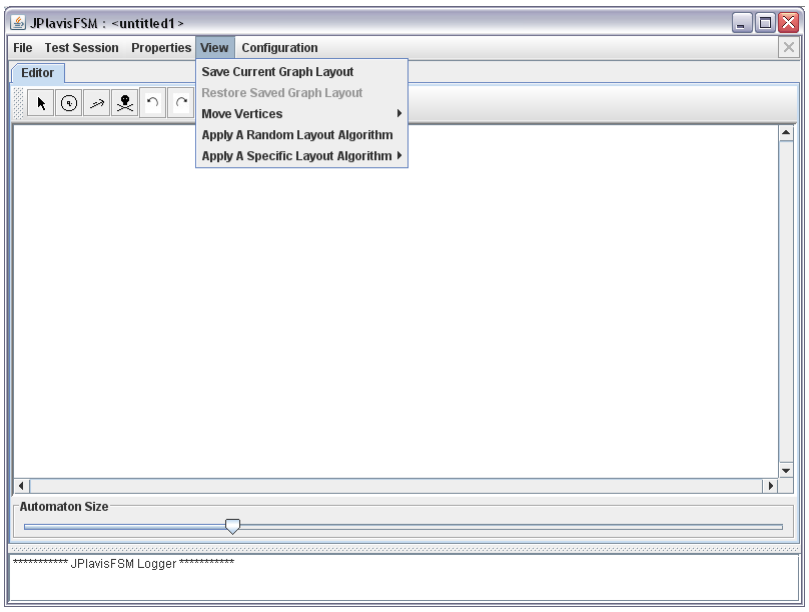

Figura 3.4: JPlavisFSM - Menu *View*.

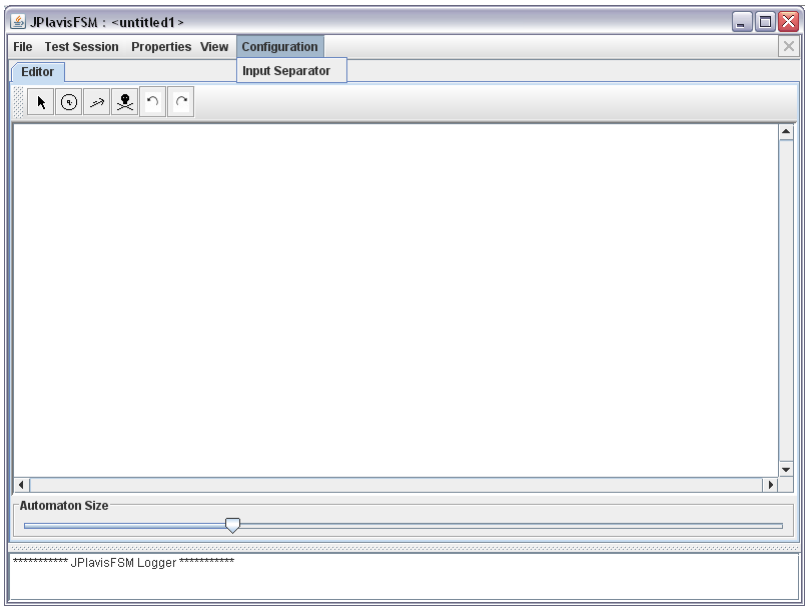

Figura 3.5: JPlavisFSM - Menu *Configuration*.

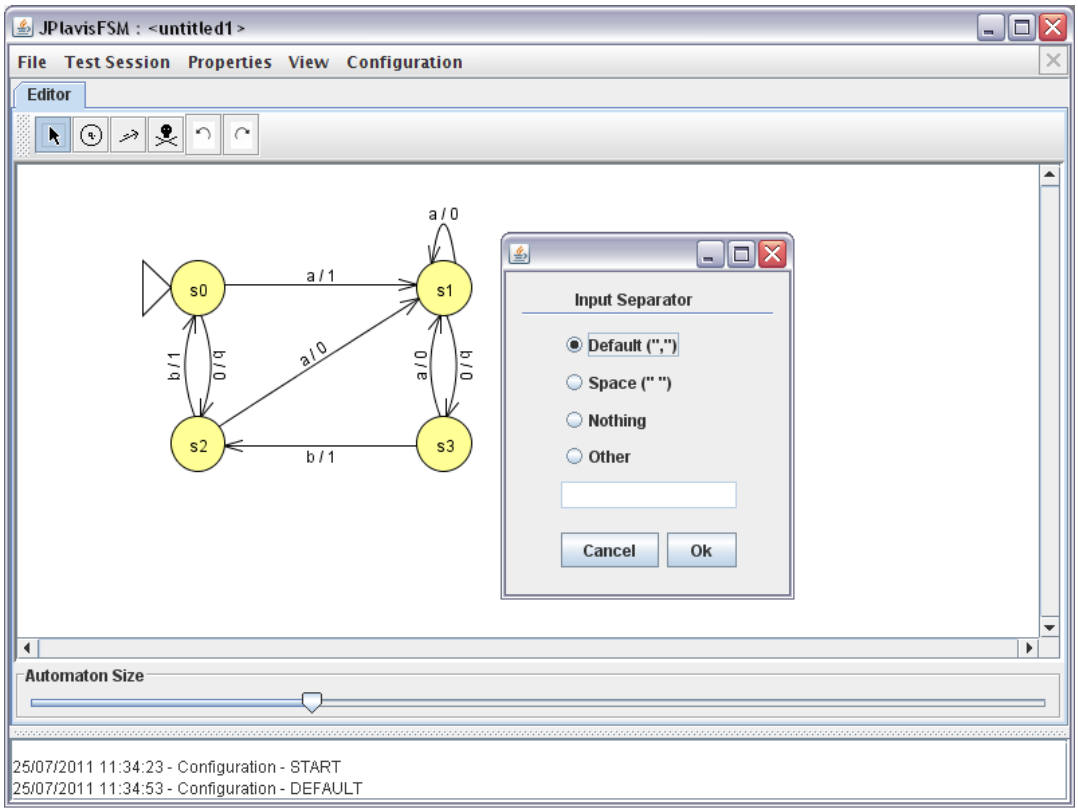

Figura 3.6: JPlavisFSM - Separador de entradas.

Capítulo

4

Edição de MEFs

Esta seção descreve como utilizar a tela de edição (Figura 4.1) de MEFs para construir o modelo a ser testado.

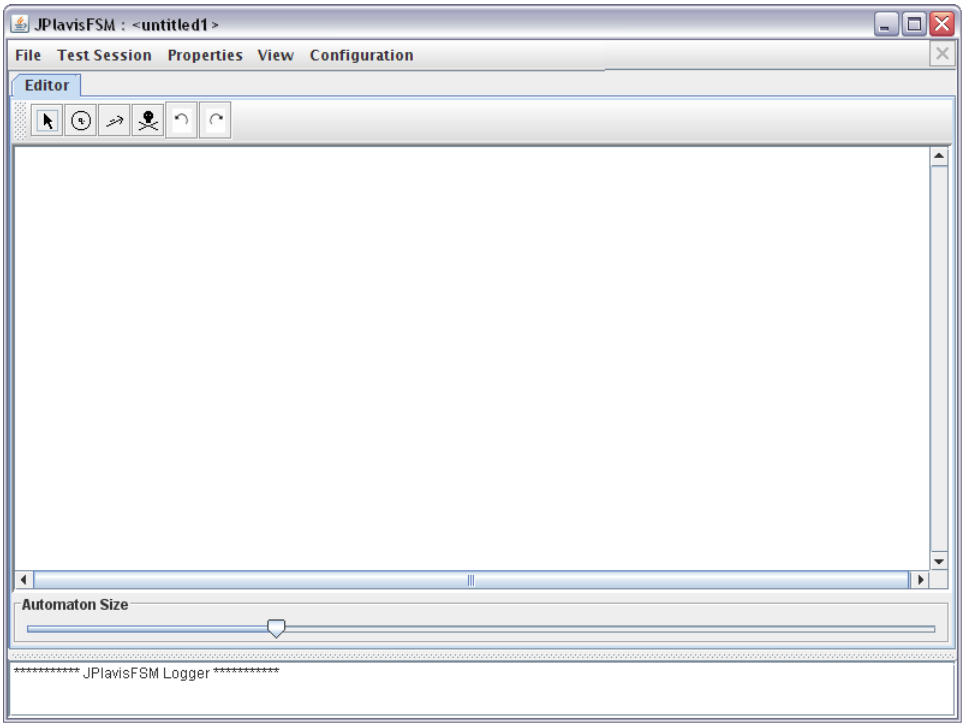

Figura 4.1: Tela inicial da JPlavisFSM - Edição de MEFs.

4.1 Criar Estados de uma MEF

Para criar uma MEF, primeiro, é necessário que sejam criados os estados. Para isso, utilize o segundo botão da aba *Editor* (tecla de atalho: s) e clique na área de edição, como é apresentado na Figura 4.2. Enquanto o botão de criação de estados estiver selecionado, a cada novo clique na área de edição um novo estado será gerado. Repita esta operação até que todos os estados necessários para representar o modelo sejam criados.

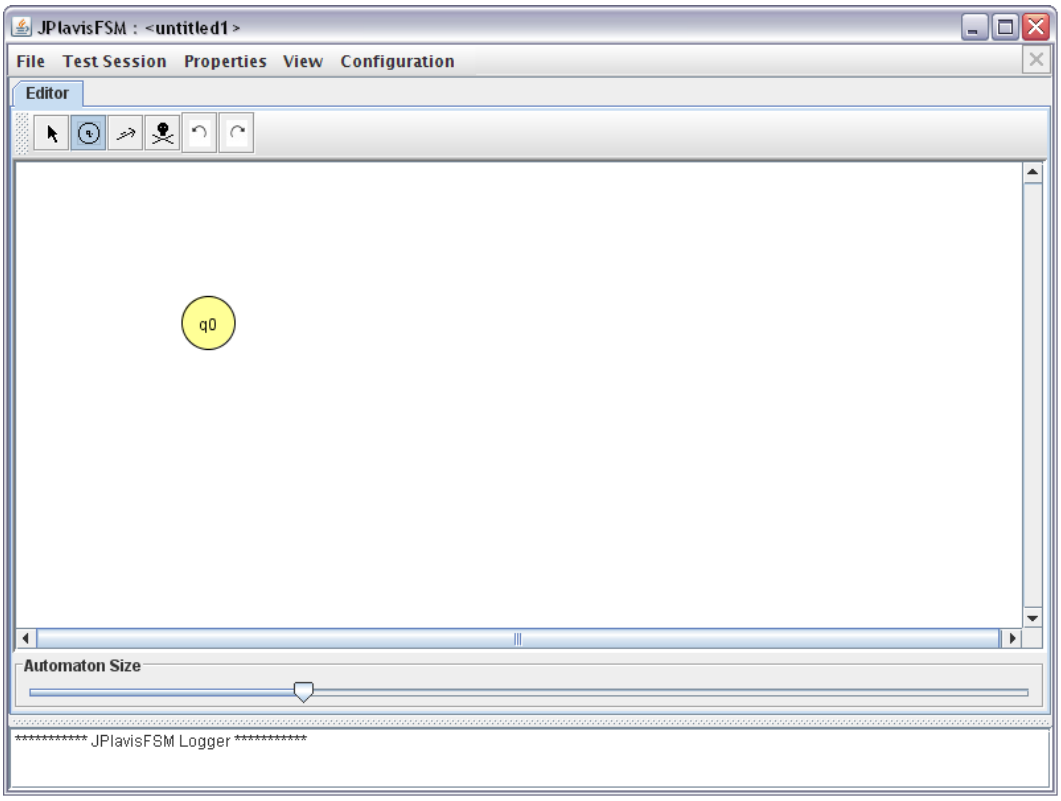

Figura 4.2: Criação de estados.

4.2 Alterar Nome de um Estado

Caso queira renomear os estados, selecione o primeiro botão da aba *Editor* (tecla de atalho: a) e clique sobre o estado escolhido com o botão direito do mouse (Figura 4.3). Selecione a opção *Set Name* e então diga qual o novo nome que deseja dar ao estado. Caso deixe o campo em branco e pressione *OK*, o estado assumirá seu nome inicial (quando foi criado).

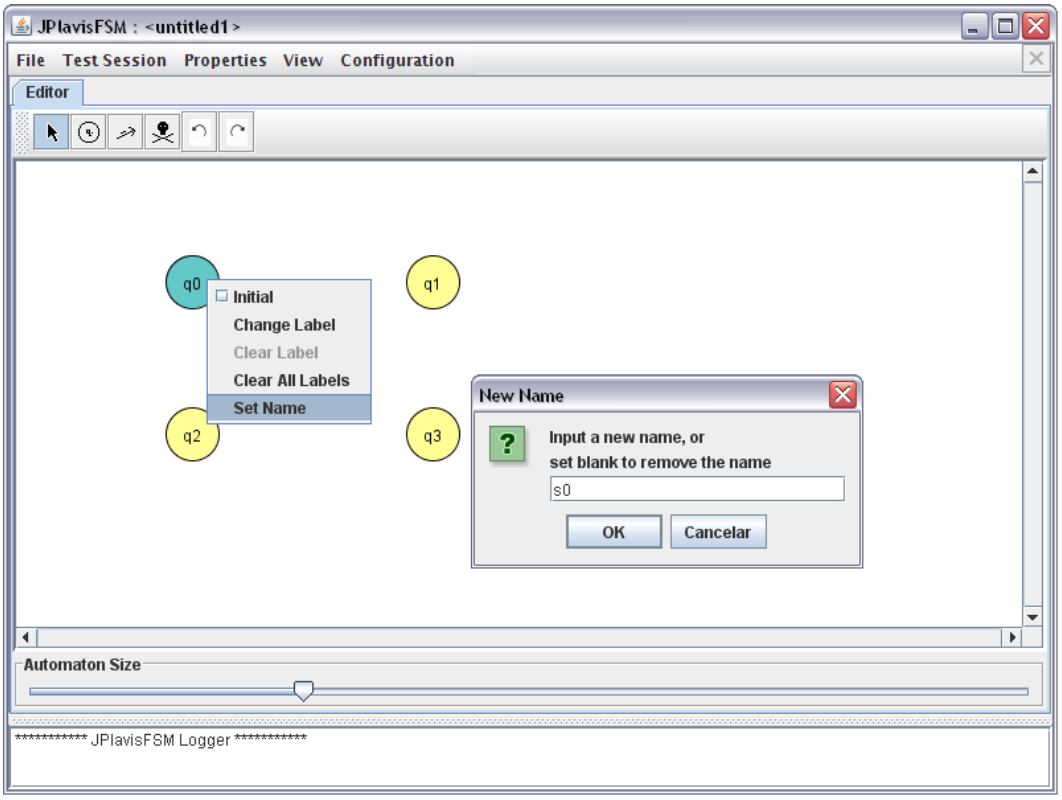

Figura 4.3: Alterando nome de um estado.

4.3 Informar Estado Inicial da MEF

Com o primeiro botão da aba *Editor* selecionado (tecla de atalho: a), clique sobre o estado escolhido com o botão direito do mouse e selecione a opção *Initial* (Figura 4.4). Apenas um estado por MEF assumirá essa opção, sendo assinalado com um marcador (seta branca) de estado inicial a sua esquerda.

4.4 Criar Transições Entre os Estados

Para criar uma transição entre dois estados, selecione o terceiro botão da aba *Editor* (tecla de atalho: t). Com o botão esquerdo do mouse, clique e segure o botão sobre o estado de origem da transição, arraste até o estado destino e solte o botão (Figura 4.5). Dois campos de edição serão exibidos: o primeiro corresponde ao símbolo de entrada da transição; e o segundo, ao símbolo de saída. Digite no primeiro campo a entrada desejada, pressione a tecla TAB e, então informe qual a saída da transição, finalizando com a tecla ENTER (Figura 4.6).

Caso deseje criar uma transição em que o estado origem e o estado destino correspondem ao mesmo estado, basta clicar sobre o estado que uma 'auto-transição' será criada.

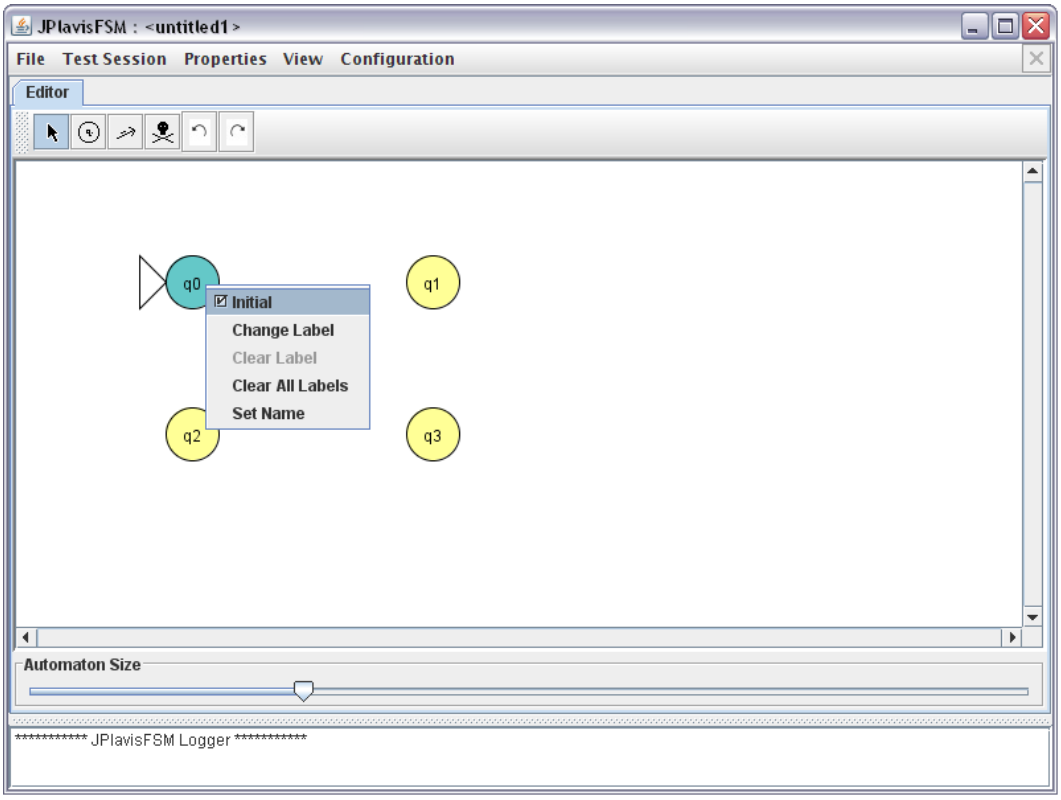

Figura 4.4: Marcando o estado inicial da MEF.

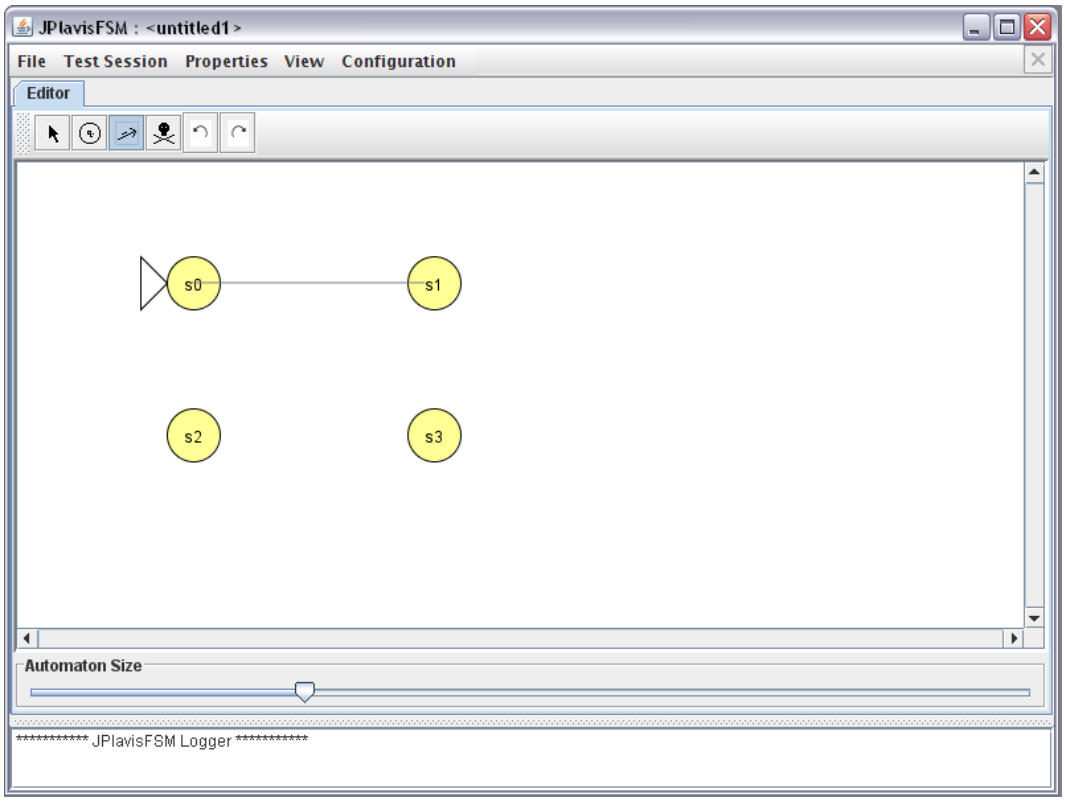

Figura 4.5: Criando uma transição entre dois estados.

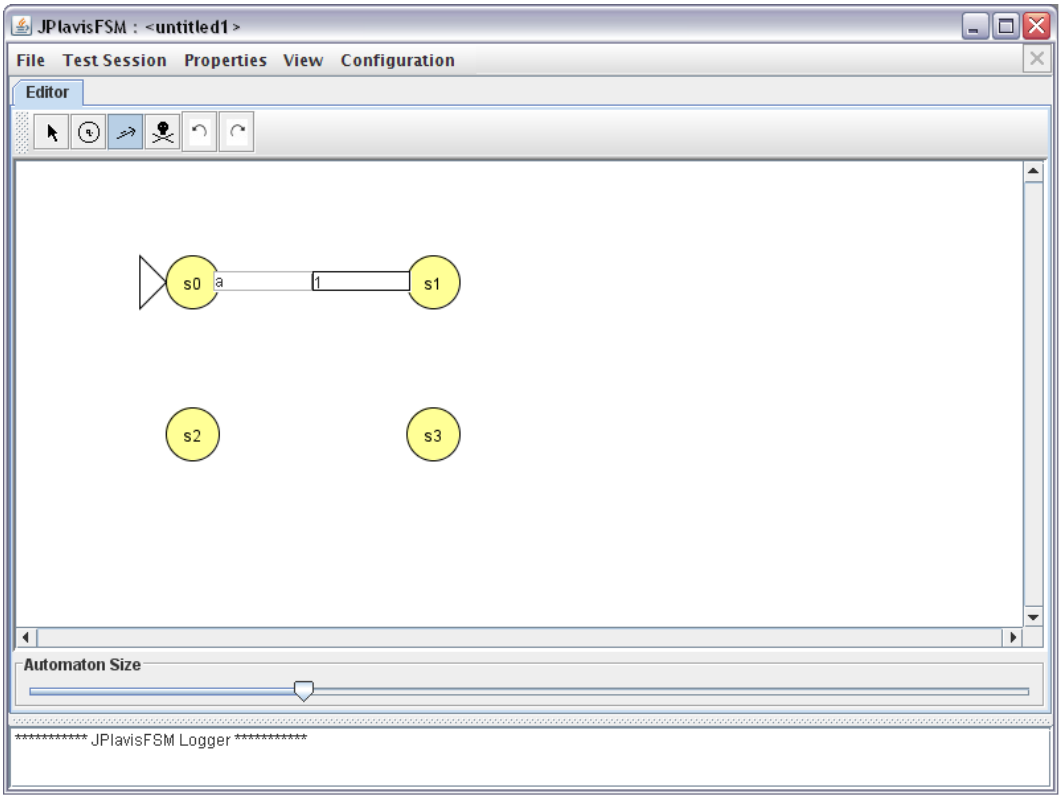

Figura 4.6: Informando a entrada/saída da transição.

Repetindo os passos de acordo com objetivo final, temos a MEF exemplo apresentada na Figura 4.7.

Após a edição, basta salvar a MEF em *File* → *Save* (Figura 4.8). A ferramenta salvará um arquivo em formato próprio, chamado 'JFLAP 4 File' (.jff). Desta forma, a MEF pode ser recuperada posteriormente para que sejam realizados novos testes.

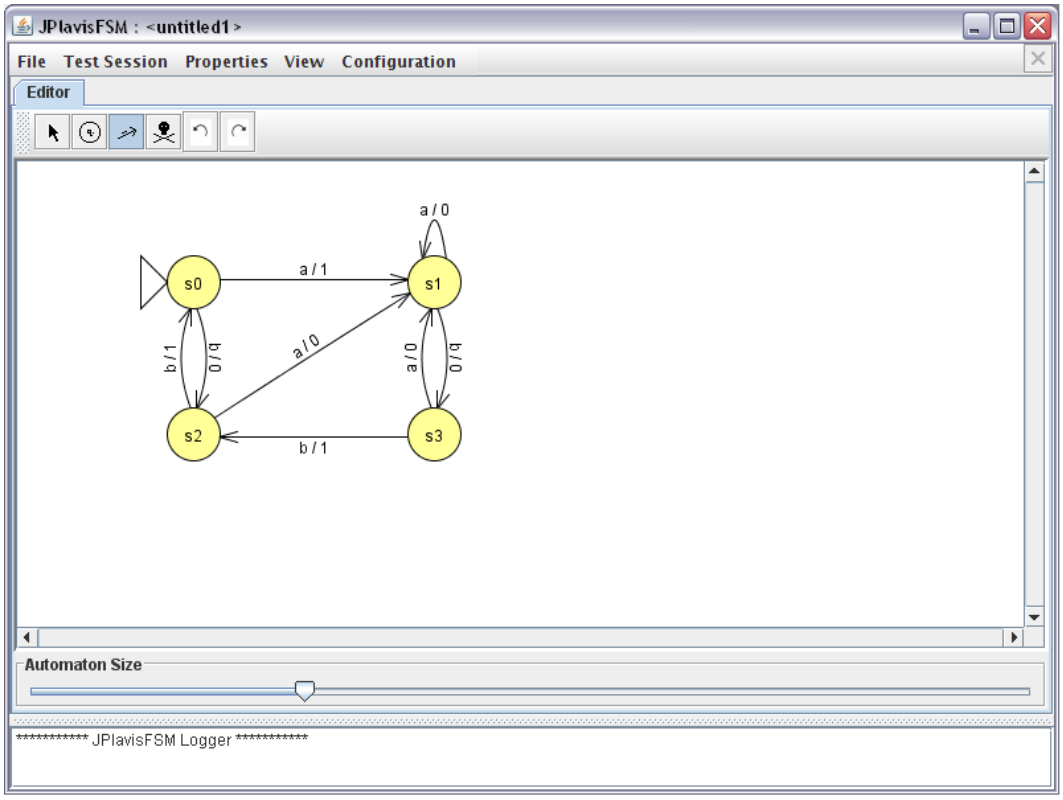

Figura 4.7: MEF final.

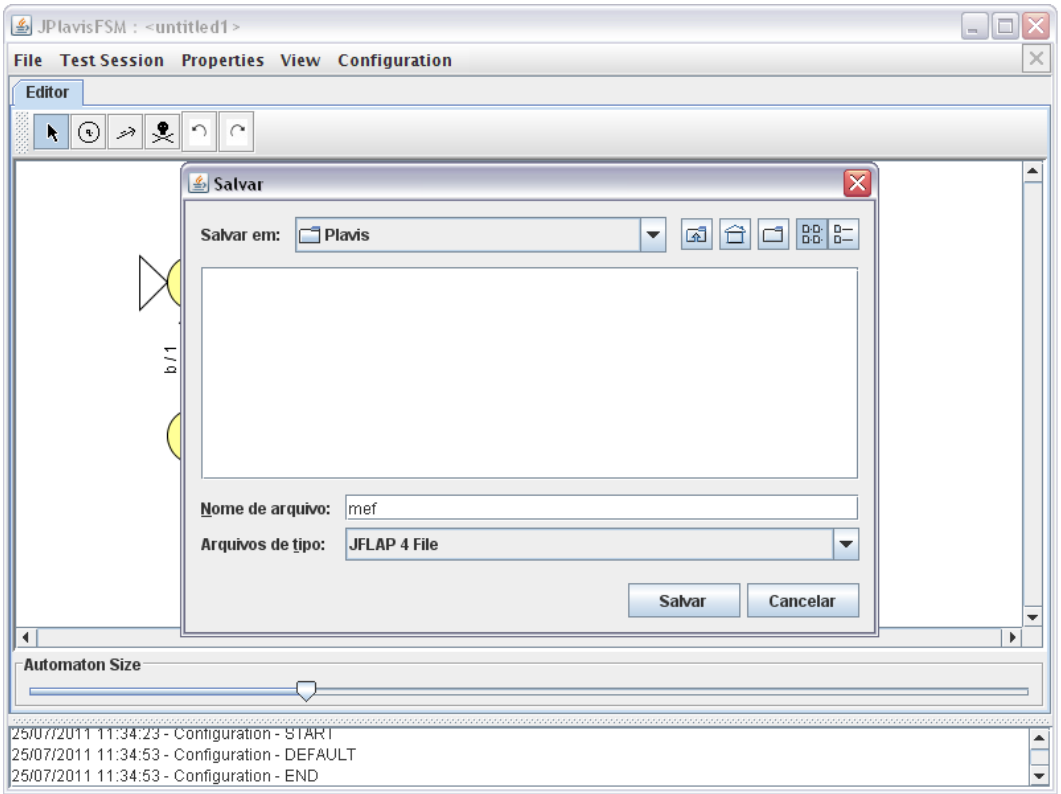

Figura 4.8: Salvando uma MEF.

Capítulo

5

Sessão de Teste

Nesta seção são apresentadas as funcionalidades disponibilizadas nas sessões de teste, bem como as funcionalidades de criar nova sessão de teste (*Test Session* → *New*) (Figura 5.1) e abrir uma sessão existente (*Test Session* → *Open*).

5.1 Criar sessão de teste

Ao selecionar a opção de criar uma nova sessão de teste, a ferramenta solicita ao usuário o local onde deseja salvar a sessão e o nome do arquivo (Figura 5.2). SUGESTÃO: Salve a sessão de teste com um nome que a relacione com a MEF em teste, por exemplo 'ts1_mef' (uma vez que a MEF em teste foi salva como mef.jff). Não é preciso adicionar extensão ao arquivo, a ferramenta salva em um formato próprio, denominado '*Plavis Test File*' (.ptf). Na Figura 5.3 é apresentado o ambiente de uma sessão de teste da JPlavisFSM.

Nesta aba estão implementadas algumas funcionalidades que guiam a atividade de teste. A esquerda está a representação da MEF em teste. A direita, a tabela de casos de teste, na qual é apresenta a sequência de entrada na primeira coluna e a respectiva saída na segunda coluna. No canto inferior direito estão disponíveis duas barras de botões, cujas funções serão explicadas a seguir.

OBSERVAÇÃO: É possível entrar manualmente com dados de teste na primeira coluna da tabela, uma vez que seja respeitada a configuração dos separadores de entrada e as entradas sejam digitadas corretamente.

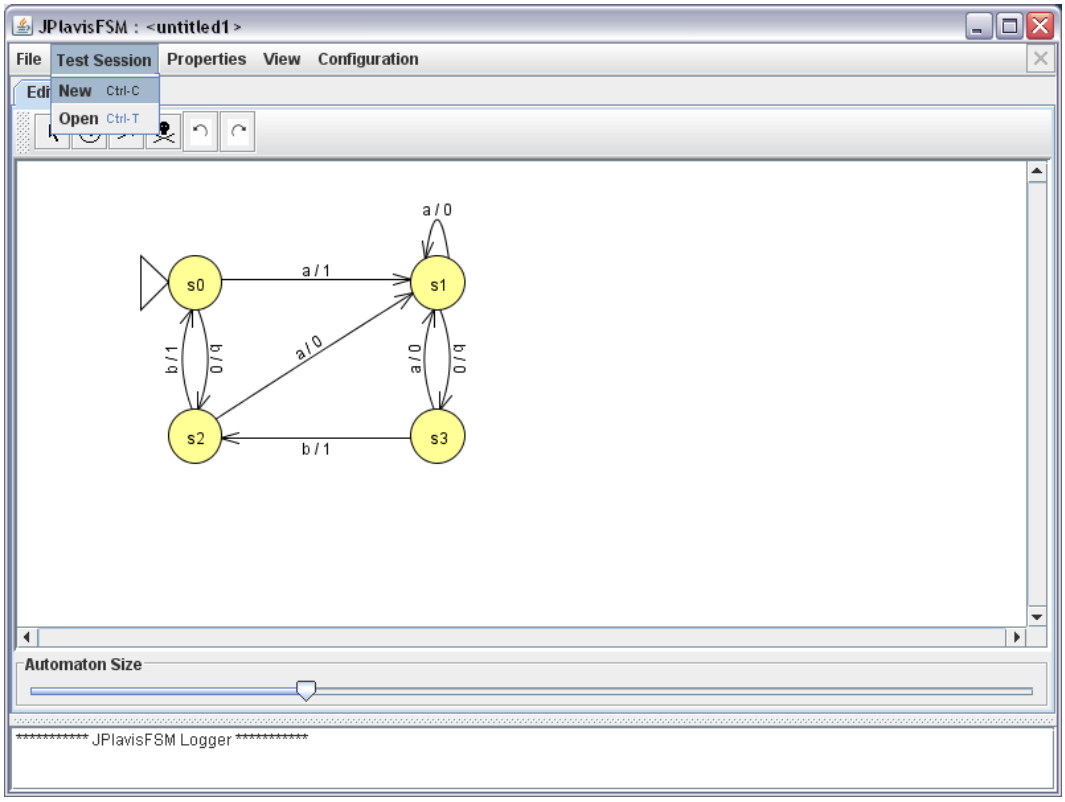

Figura 5.1: JPlavisFSM - Sessão de Teste

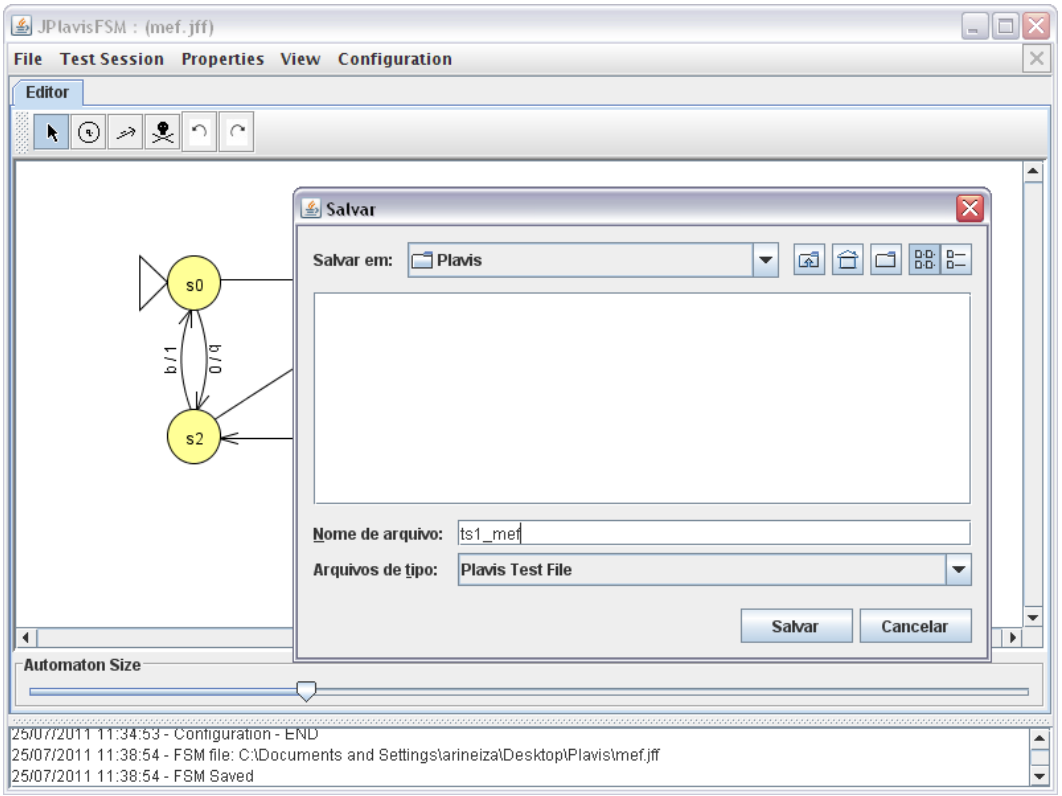

Figura 5.2: JPlavisFSM - Criar Sessão de Teste

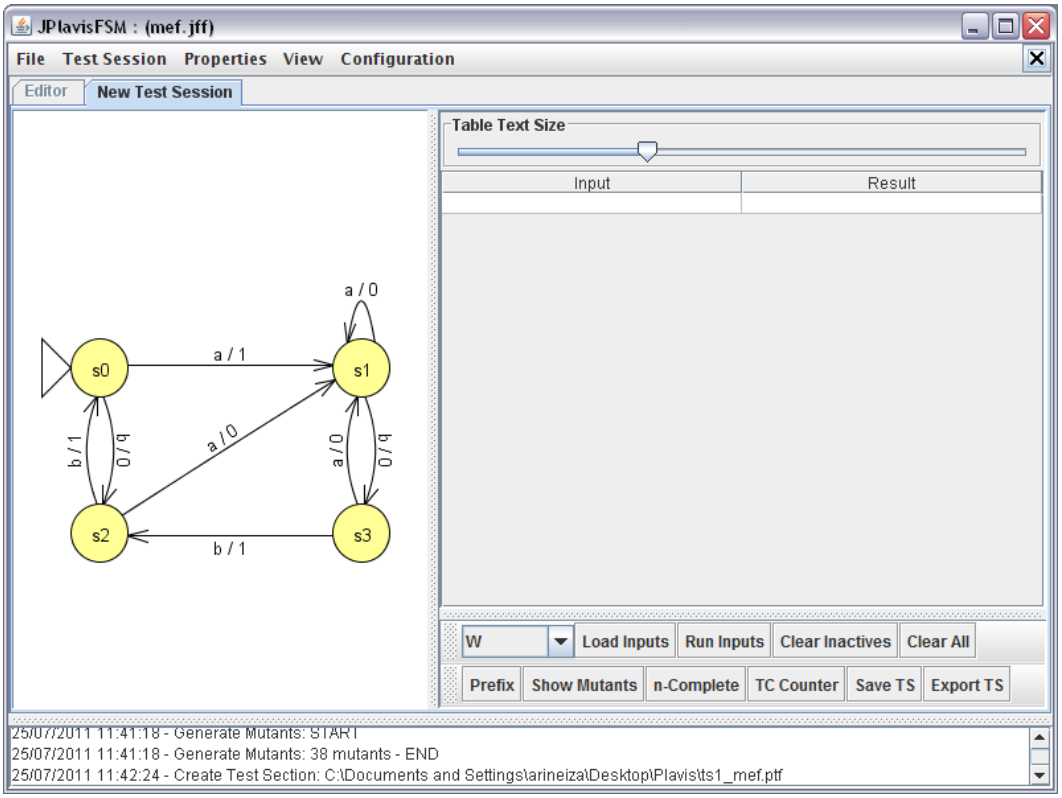

Figura 5.3: JPlavisFSM - Aba *New Test Session*.

5.2 Load Inputs

O botão *Load Inputs* está vinculado ao ComboBox que está a sua esquerda. Com esse botão é possível importar casos de teste a partir: (1) do método W, já implementado na ferramenta; (2) *From file*, que importa os casos de teste de um arquivo de texto qualquer; (3) de algum método externo.

5.2.1 Métodos de Geração

A ferramenta disponibiliza cinco métodos de geração: W, UIO, HSI, SPY e Switch-Cover. Para carregar os casos de teste gerados pelos métodos implementado na ferramenta, basta selecionar a opção desejada no ComboBox (Figura 5.4) e clicar em *Load Inputs* (Figura 5.5).

Como o método W somente é aplicável a MEFs completamente especificadas, há a opção de auto-completar a MEF. A funcionalidade "auto-complete" insere *self-loops* nos estados parcialmente especificados, o que corresponde a transições com origem e destino nos estados com entradas faltantes. Para cada entrada faltante, é gerada uma transição com saídas vazias, simbolizadas pela palavra "*Empty*". Desta forma, a MEF é completada artificialmente sem perda de alterações semânticas, ou seja, o comportamento da

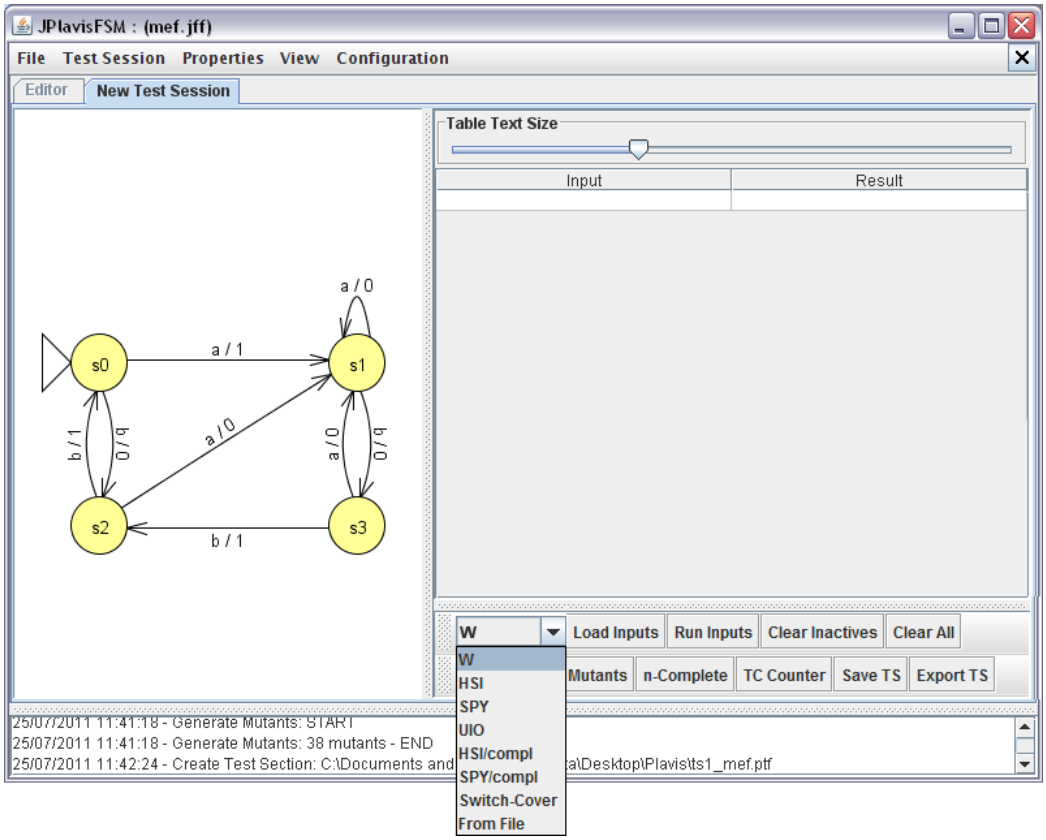

Figura 5.4: JPlavisFSM - *Métodos de Geração.*

nova MEF continua equivalente a MEF original. Há também as opções: "HSI-compl" e "SPY-compl", que é a aplicação da funcionalidade auto-complete antes da execução dos métodos HSI e SPY, respectivamente.

OBSERVAÇÃO: Na versão atual da ferramenta, os métodos HSI, SPY, UIO e Switch Cover estão disponíveis apenas para ambiente Linux.

5.2.2 From File

Para importar casos de teste a partir de algum arquivo de texto é necessário seguir algumas regras (Figura 5.6).

1. Na primeira linha do arquivo deve estar descrito qual o separador de entradas que deve ser utilizado para ler as sequências de teste:
   - DEFAULT, para vírgula;
   - SPACE, para espaço em branco;
   - NOTHING, quando deve-se considerar entradas como caracteres;
   - ou o próprio símbolo do separador para o caso de não ser nenhum dos descritos anteriormente)

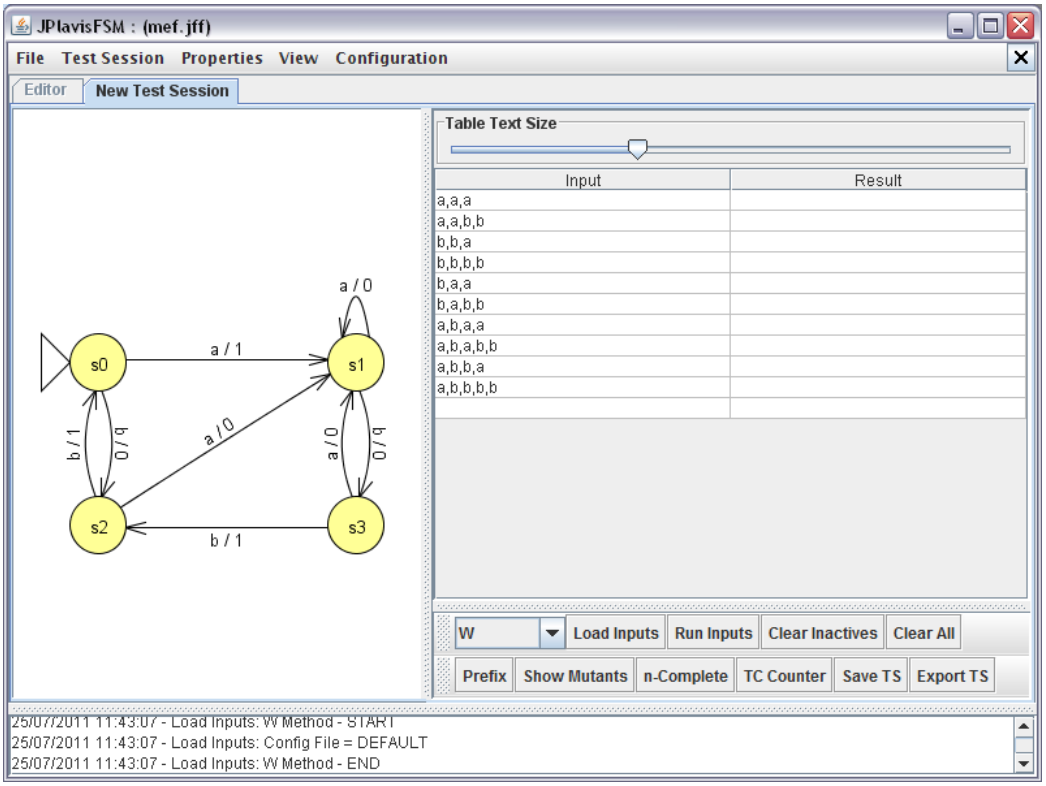

Figura 5.5: JPlavisFSM - *Load Inputs*.

2. Nas demais linhas, as sequências de teste (uma por linha).

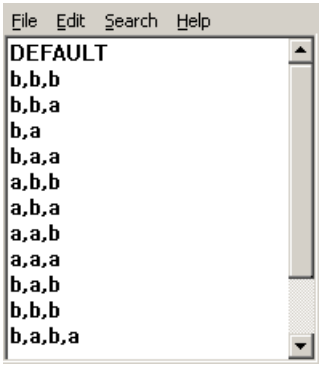

Figura 5.6: Formato de arquivo para importar casos de teste

5.2.3 Método Externo

Se a opção for importar casos de teste a partir de um método externo, ele deve gerar um arquivo de texto com as sequências geradas da mesma forma que apresentado na Figura 5.6. O método externo deve ser colocado em uma pasta *Methods* junto ao executável da ferramenta (Figura 5.7).

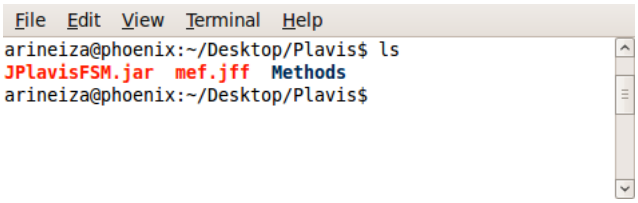

Figura 5.7: Formato de arquivo para importar casos de teste

A ferramenta reconhece automaticamente o novo método colocado nesta pasta. O método deve ser desenvolvido em Java (ou seja, aceita executáveis .jar). A JPlavisFSM executa uma chamada externa seguindo o padrão: **java -jar *newMethod*.jar fsm.txt**.

O formato do arquivo de entrada – fsm.txt – segue o padrão:

estado origem - - entrada / saída -> estado destino

Por exemplo: MEF com 3 estados, entradas a e b, saídas 0 e 1, com estado inicial s0, completamente especificada, determinística, inicialmente conexa e minimal.

s0 - - a / 0 -> s1
s0 - - b / 1 -> s2
s1 - - a / 0 -> s2
s1 - - b / 0 -> s1
s2 - - a / 1 -> s0
s2 - - b / 0 -> s1

O estado inicial é determinado pelo estado origem da primeira transição do arquivo de especificação da MEF, ou seja, no exemplo acima corresponde a $s0$. O arquivo de saída deve ser gerado com nome de fsmOUT.txt e respeitar o formato descrito na Seção 5.2.2. A ferramenta JPlavisFSM, ao executar o novo método implementado, verifica se o conjunto gerado é completo[1].

5.3 Run Inputs

Para calcular as saídas para as sequências de teste listadas na tabela, é necessário pressionar o botão *Run Inputs*. Ao executar as entradas, também é calculado o score de mutação, como pode ser observado no *log* na área inferior da ferramenta (Figura 5.8).

[1]Esta funcionalidade ainda está limitada para o caso de entradas como caracteres

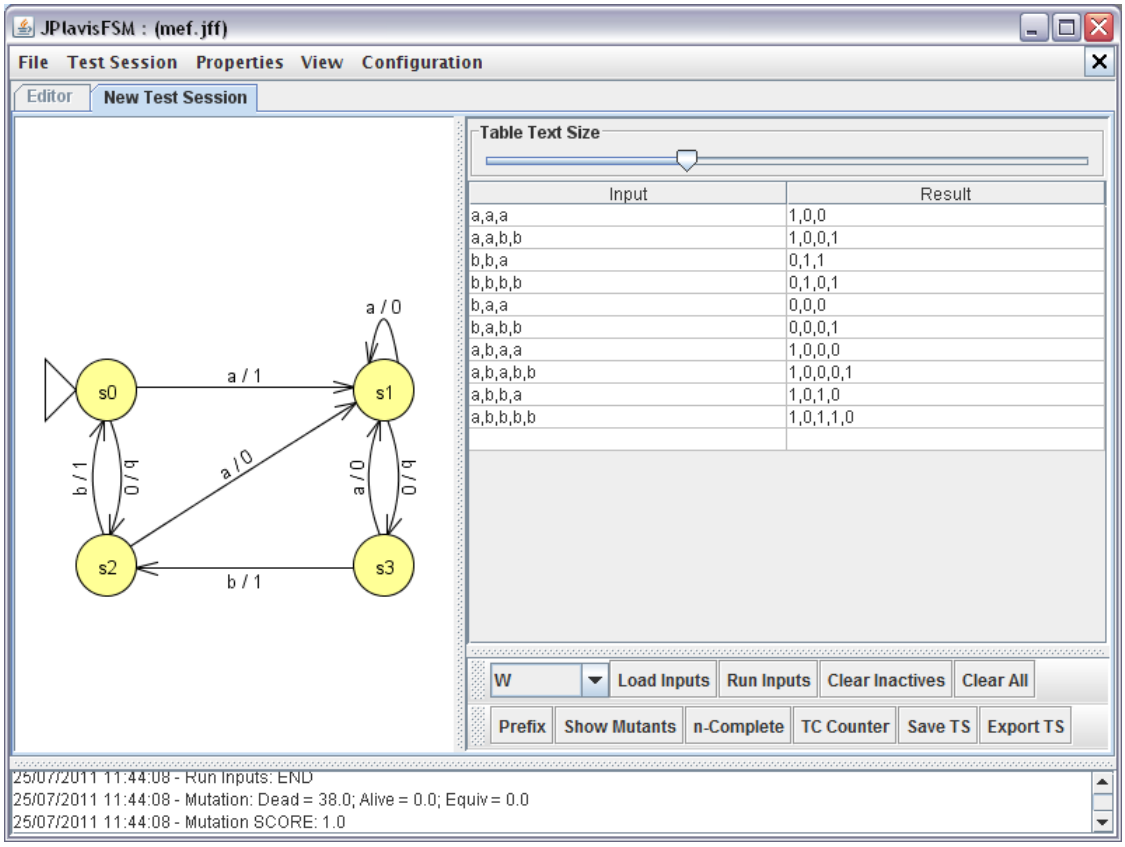

Figura 5.8: JPlavisFSM - *Run Inputs*.

5.4 Prefix

O botão *Prefix* marca como inativo os casos de teste que correspondam ao prefixo de outras sequências ou à redundâncias (casos de teste idênticos). Um caso de teste inativo é sinalizado por um um '#' no início da linha.

Por exemplo, na Figura 5.9 é exibido a tabela de entradas com dois casos de teste identicos (primeira e última linha). Ao pressionar o botão "Prefix", o caso de teste da primeira linha é marcado como inativo (Figura 5.10). Quando re-executados os casos de teste, apenas a linha ativa tem a sua saída calculada (Figura 5.11).

5.5 Clear Inactives e Clear All

Existem duas opções para limpar a tabela implementadas: (1) eliminar os testes inativos (*Clear Inactives*) e (2) eliminar todos os testes (*Clear All*). No exemplo da Figura 5.12, foram marcados como inativas a primeira e a segunda sequência de teste. Ao pressionar o botão *Clear Inactives*, os testes marcados como inativos são apagados (Figura 5.13).

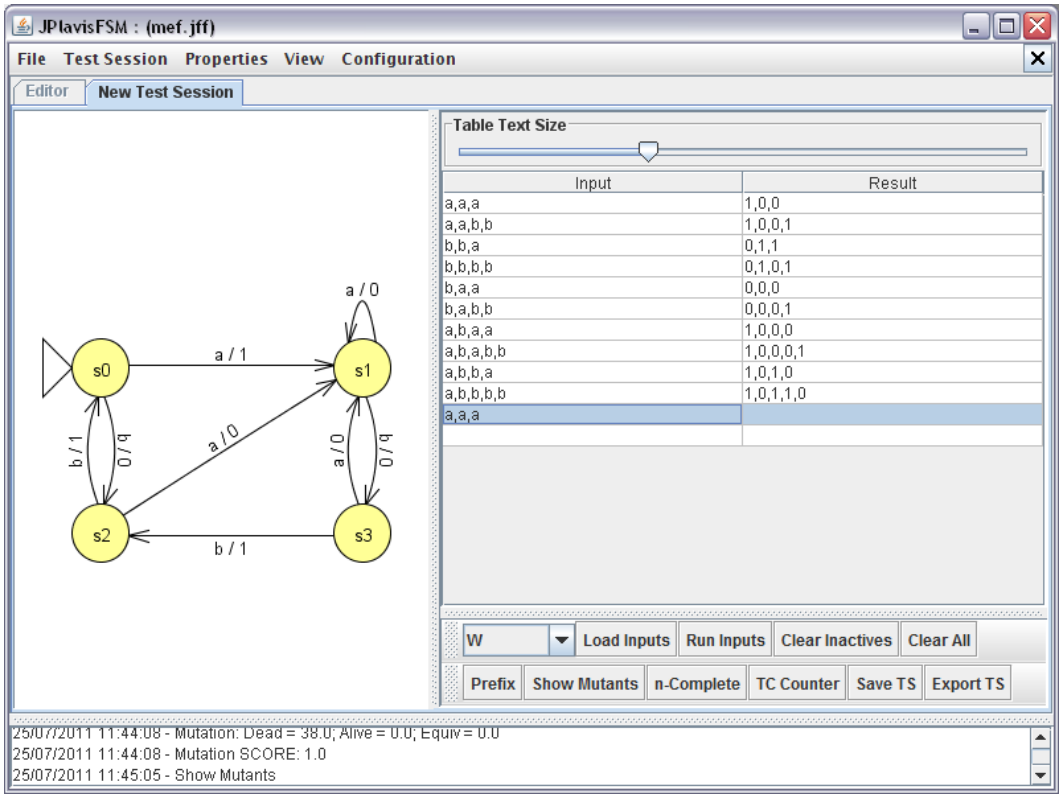

Figura 5.9: JPlavisFSM - Nova sequência de teste.

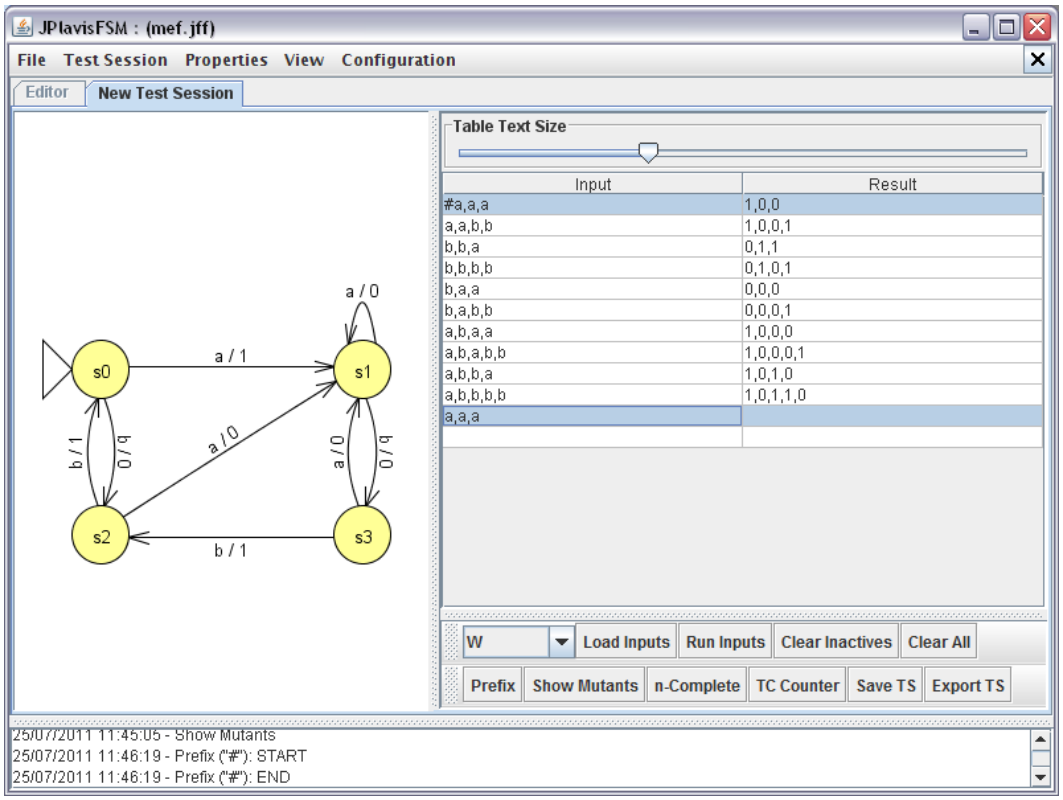

Figura 5.10: JPlavisFSM - *Prefix*.

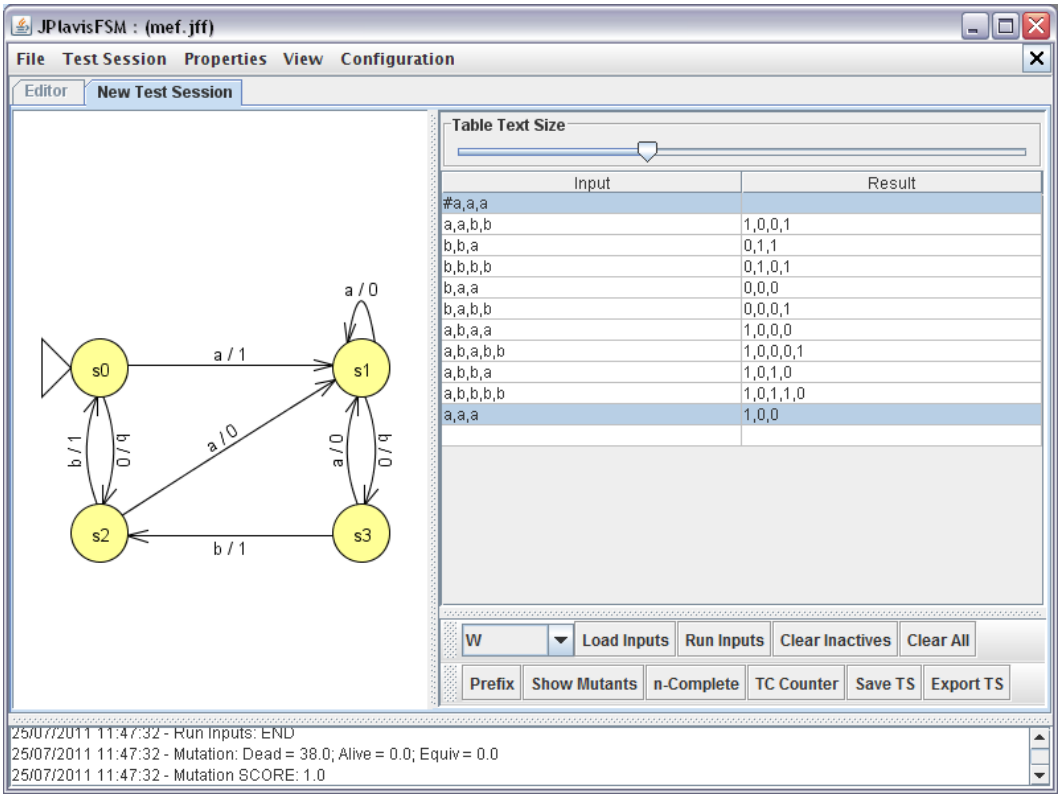

Figura 5.11: JPlavisFSM - Execução dos testes.

OBSERVAÇÃO: uma sequência pode ser marcada manualmente como inativa, clicando duas vezes na linha desejada e digitando um '#' no início da sequência.

5.6 Show Mutants

Para visualizar os mutantes gerados, ao clicar no botão *Show Mutants*, é aberta uma nova aba que exibe os mutantes, seus *status* e o score de mutação.

Ao terminar de visualizar os mutantes, fechar a aba para voltar a aba da sessão de teste.

5.7 Save TS

O botão *Save TS* atualiza o arquivo gerado para armazenar tanto o conteúdo da sessão de teste quanto o status dos mutantes. Ao salvar uma sessão de teste, os espaços em branco da tabela são suprimidos.

ATENÇÃO: Sempre salvar a sessão de teste antes de finalizá-la, pois os dados não são salvos automaticamente! E é possível fechar uma sessão de teste sem salvá-la antes.

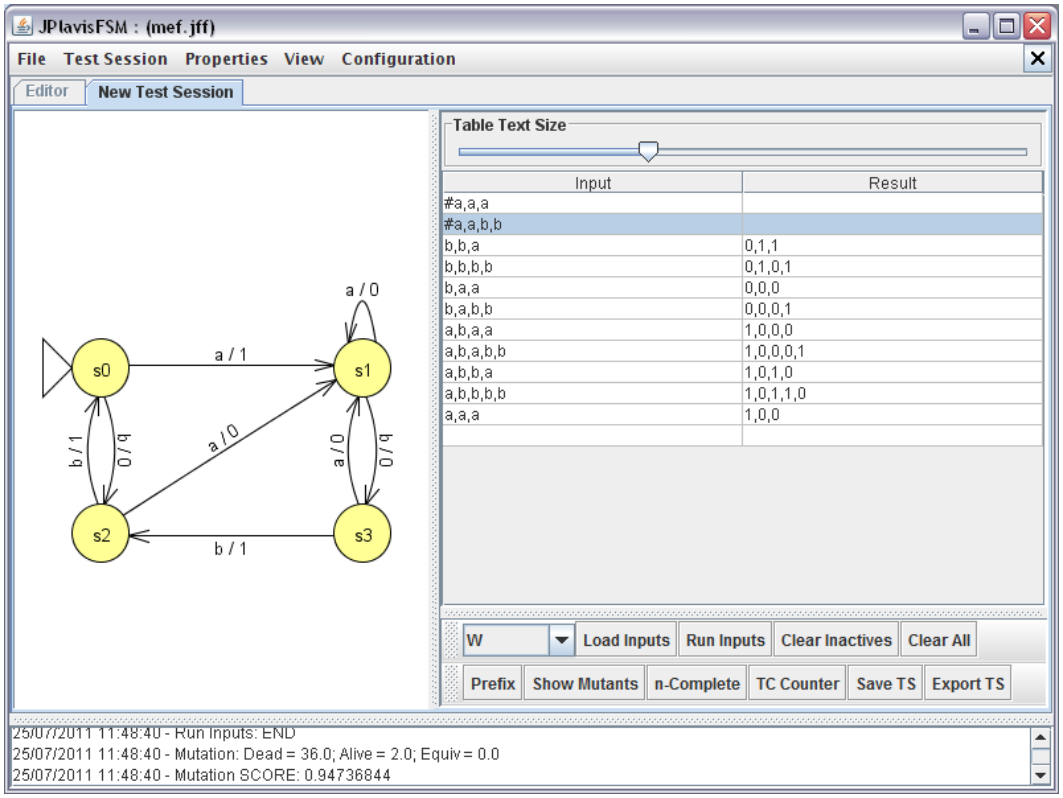

Figura 5.12: JPlavisFSM - Marcando testes como inativos.

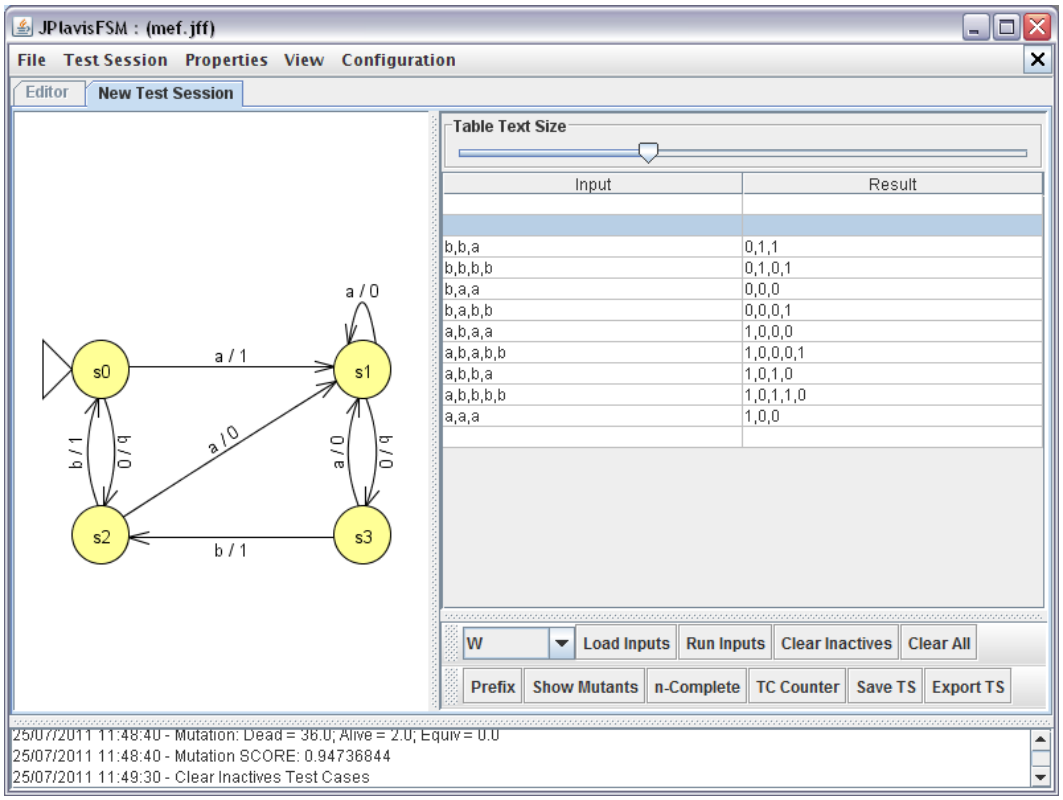

Figura 5.13: JPlavisFSM - *Clear Inactives*.

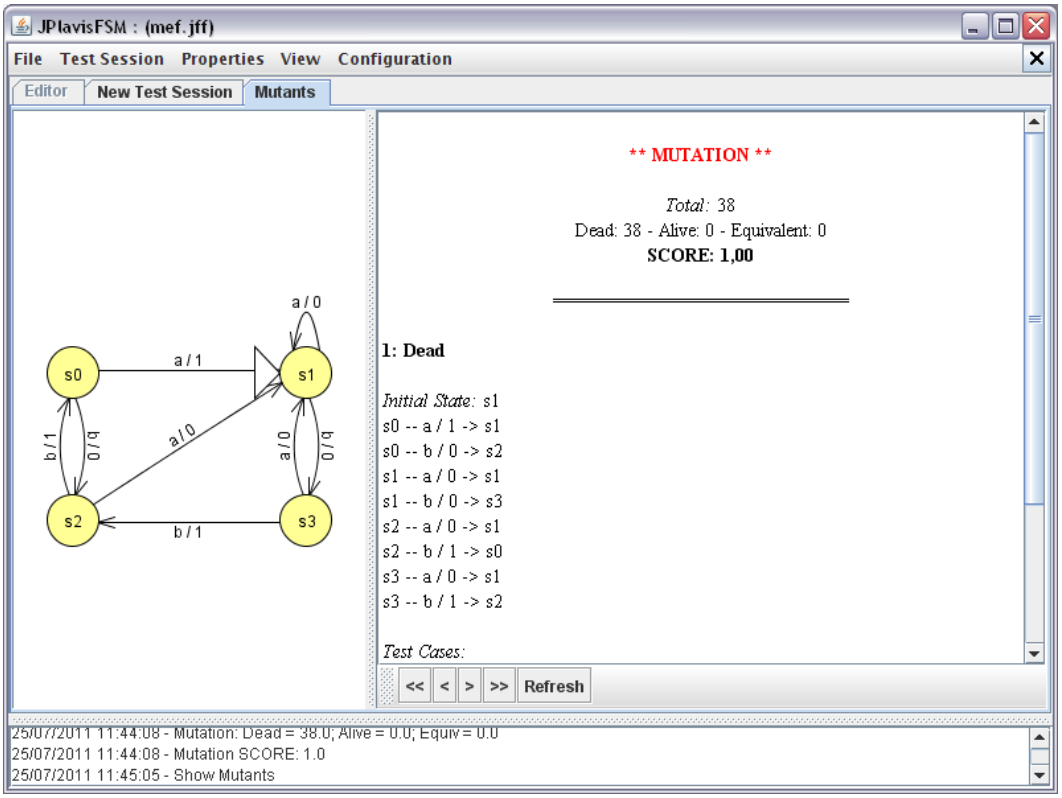

Figura 5.14: JPlavisFSM - Aba *Mutants*.

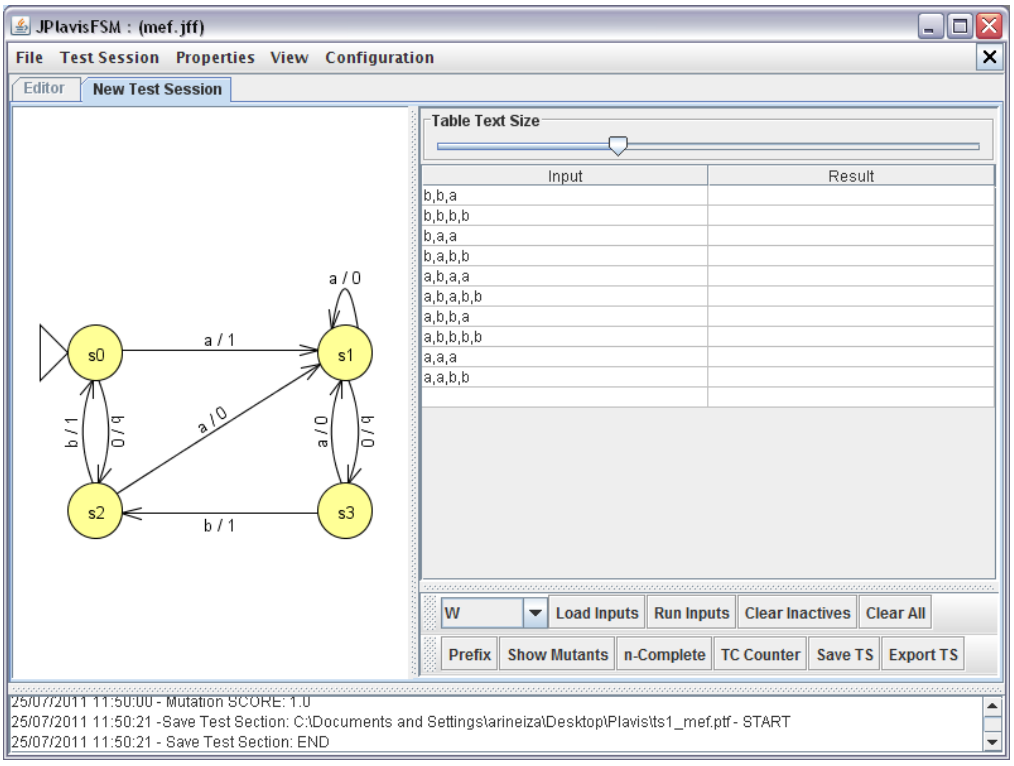

Figura 5.15: JPlavisFSM - *Save TS*.

5.8 Count TC

O botão *Count TC* exibe o número de sequências de teste contidos na tabela, apresentando o número total de testes, e a respectiva quantidade de testes ativos e inativos (Figura 5.16).

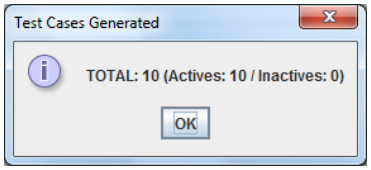

Figura 5.16: JPlavisFSM - Resultado do *Count TC* aplicado a Figura 5.15.

5.9 Export TS

O botão *Export TS* exporta a sessão de teste aberta, criando um arquivo texto na pasta onde se encontra o executável da ferramenta com os testes presentes na tabela e suas respectivas saídas (caso tenham sido geradas).

5.10 n-Complete

O botão n-Complete verifica se o conjunto de teste apresentado na tabela é completo [2] (Figura 5.8).

OBSERVAÇÃO: Na versão atual da ferramenta, esta funcionalidade pode ser executada apenas em ambiente Linux.

OBSERVAÇÃO 2: Para Habilitar a ferramenta n-complete na JPlavisFSM, é necessário:

1. ir até o diretório raiz da ferramenta (onde o .jar está);
2. Entrar no diretório *Methods*;
3. Escolher qual versão da ferramenta utilizar (32 ou 64 bits) (ver qual arquitetura é o seu computador);
4. Renomear a versão desejada para **n-complete**;
5. Remover a outra versão da ferramenta (não utilizada);

[2] Ver último parágrafo da Seção 2.3

6. Executar os comandos: *chmod +x n-complete* e *chmod +x script.sh*, para habilitar a ferramenta.

Na Figura 5.17 é ilustrado o procedimento descrito acima.

```
1 ~$ cd JPlavisFSM/
2 ~/JPlavisFSM$ ls
3 JPlavisFSM.jar  ManualJPlavisFSM.pdf  Methods
4 ~/JPlavisFSM$ cd Methods/
5 ~/JPlavisFSM/Methods$ ls
6 n-complete32  n-complete64  script.sh
7 ~/JPlavisFSM/Methods$ mv n-complete64 n-complete
8 ~/JPlavisFSM/Methods$ ls
9 n-complete  n-complete32  script.sh
10 ~/JPlavisFSM/Methods$ rm n-complete32
11 ~/JPlavisFSM/Methods$ ls
12 n-complete  script.sh
13 ~/JPlavisFSM/Methods$ chmod +x n-complete
14 ~/JPlavisFSM/Methods$ chmod +x script.sh
15 ~/JPlavisFSM/Methods$
```

Figura 5.17: Habilitando a ferramenta n-Complete.

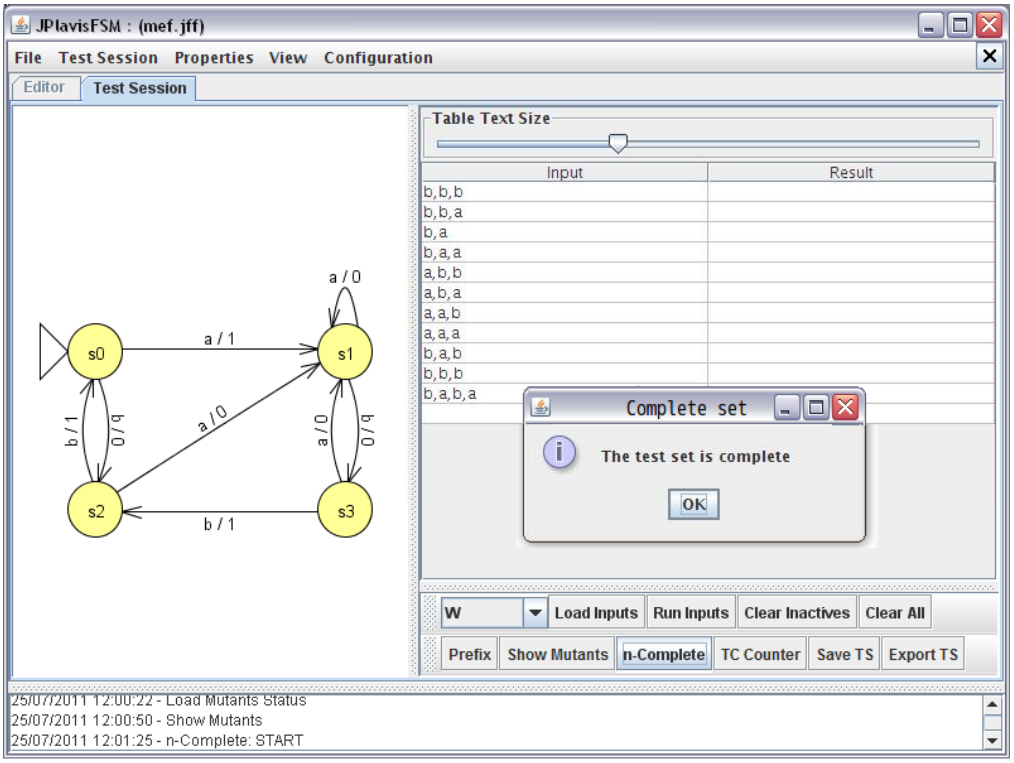

Figura 5.18: JPlavisFSM - *n-Complete*.

5.11 Abrir Sessão de Teste

Se o objetivo é retomar uma sessão de teste anteriormente salva, basta escolher a opção *Test Session* → *Open*. Serão listados apenas os arquivos gerados pela ferramenta (Figura

5.19), pois toda sessão de teste salva gera um arquivo que armazena o status dos mutantes, evitando assim a necessidade de processar novamente todos os mutantes.

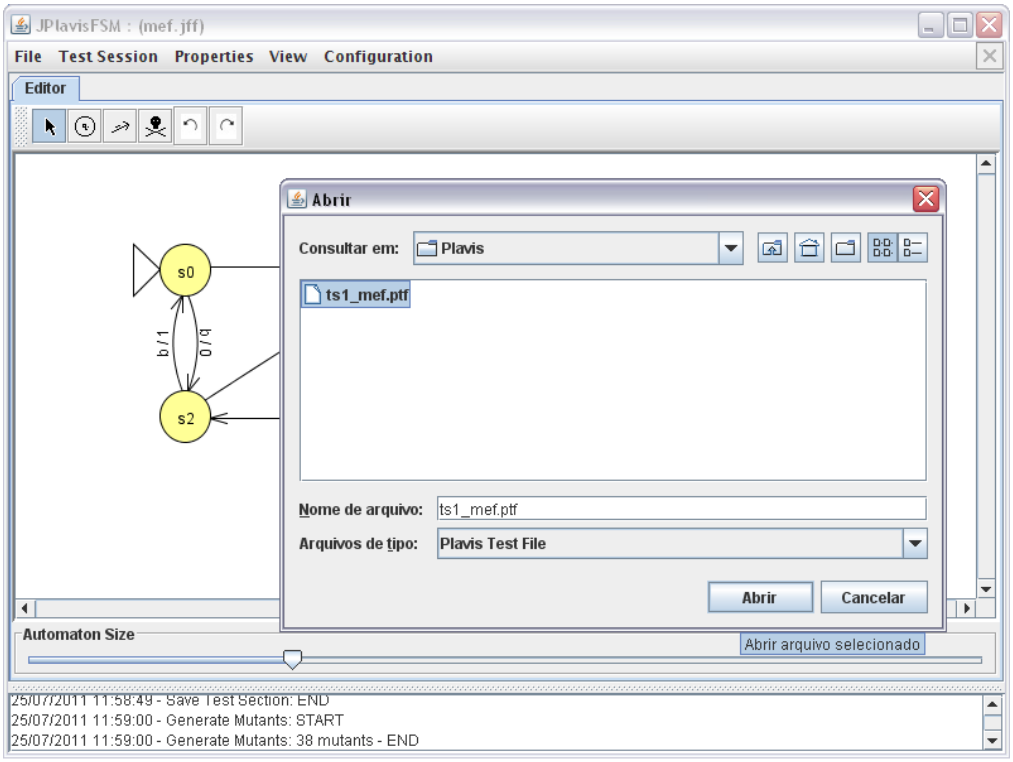

Figura 5.19: JPlavisFSM - Abrir sessão de teste.

A sessão de teste salva todos os testes que estavam na tabela, mesmo que inativos. Ao reabrir a sessão, o status de mutação anteriormente alcançado com os testes é carregado e pode ser visualizado na aba *Mutants* mesmo antes de se re-executar os testes (Figura 5.20)

ATENÇÃO: Sempre salvar a sessão de teste antes de finalizá-la, pois os dados não são salvos automaticamente! E é possível fechar uma sessão de teste sem salvá-la antes.

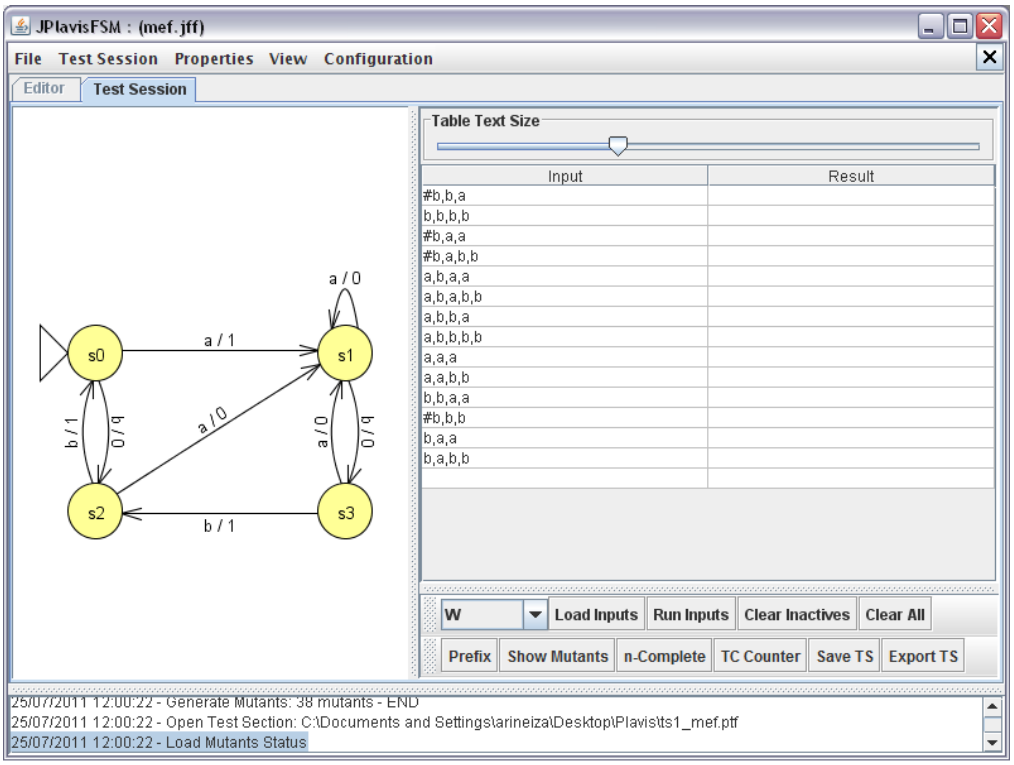

Figura 5.20: JPlavisFSM - Sessão de teste carregada.

Agradecimentos

Agradecimento às agências Capes e FAPESP pelo apoio financeiro; ao programa de pós-graduação do ICMC (USP São Carlos) e ao INCT-SEC pela estrutrura fornecida; e a Profa. Dra. Ana Maria Ambrósio pelo apoio durante a fase de desenvolvimento da ferramenta e pela utilização da JPlavisFSM em disciplinas do programa de pós-graduação do INPE.

Referências Bibliográficas

Chow, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, v. 4, p. 178 – 187, 1978.

Delamaro, M. E.; Barbosa, E. F.; Vincenzi, A. M. R.; Maldonado, J. C. Teste de mutação. In: Delamaro, M. E.; Maldonado, J. C.; Jino, M., eds. *Introdução ao Teste de Software*, Elsevier, p. 9 – 25, 2007a.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. Conceitos básicos. In: Delamaro, M. E.; Maldonado, J. C.; Jino, M., eds. *Introdução ao Teste de Software*, Elsevier, p. 1 – 7, 2007b.

Fabbri, S. C. P. F.; Maldonado, J. C.; Masiero, P. C.; Delamaro, M. E. Proteum/FSM: A tool to support finite state machine validation based on mutation testing. In: *Proceedings of the 19th International Conference of the Chilean Computer Science Society*, Washington, DC, USA: IEEE Computer Society, 1999, p. 96 – 104.

Fujiwara, S.; Von Bochmann, G.; Khendek, F.; Amalou, M.; Ghedamsi, A. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, v. 17, p. 591 – 603, 1991.

Mathur, A. P. *Foundations of software testing*, v. 1. Pearson Education, 2008.

Petrenko, A.; Yevtushenko, N.; Lebedev, A.; Das, A. Nondeterministic state machines in protocol conformance testing. In: *Proceedings of the 6th International Workshop on Protocol Test systems VI (IFIP TC6/WG6.1)*, Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, p. 363–378.

Pimont, S.; Rault, J. C. A software reliability assessment based on a structural and behavioral analysis of programs. In: *Proceedings of the 2nd international conference*

on Software engineering (ICSE76), Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, p. 486 – 491.

Sabnani, K.; Dahbura, A. A protocol test generation procedure. *Computer Networks and ISDN Systems*, v. 15, n. 4, p. 285 – 297, 1988.

Simão, A.; Petrenko, A. Checking completeness of tests for finite state machines. *IEEE Transactions on Computers*, v. 59, n. 8, p. 1023 –1032, 2010.

Vuong, S.; Chan, W.; Ito, M. The UIOv-Method for protocol test sequence generation. In: *Proceedings of the 2nd Workshop Protocol Test Systems*, 1989, p. 161 – 175.