Estudo do Teste de Mutação para a Linguagem Standard ML

Thaise Yano
Adenilso da Silva Simão
José Carlos Maldonado
Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação
São Carlos, Brasil, Caixa Postal 668 - CEP 13560-970
{tyano,adenilso,jcmaldon}@icmc.usp.br

Abstract

Functional programming languages, such as SML (Standard Meta Language), Haskell and Lisp, focus on rules and matching of patterns, in contrast to procedural languages in which programs are written as a sequence of instructions. Programs in functional languages may have errors due to the misunderstanding of their properties. Therefore, in this work, we establish mechanisms to investigate the aplicability of Mutation Testing for testing functional programs, written in SML. Mutation Testing is a test criterion that allows to evaluate the quality of a test set and to guide the generation of test sets. The existence of a tool to support this criterion is essential due to the large amount of information related to its application. The web tool PROTEUM/SML, developed with the aim of applying the Mutation Testing to SML, implements the mutation operators defined in this work.

Keywords: Mutation Testing, Functional Programming Language, Standard ML.

Resumo

Linguagens de programação funcionais, tais como SML (Standard Meta Language), Haskell e Lisp, enfatizam regras e casamento de padrões, ao contrário das linguagens procedimentais em que os programas são escritos como uma seqüência de instruções. Os programas em linguagens funcionais podem conter erros pela falta de entendimento de suas propriedades. Assim, neste trabalho, estabelecem-se subsídios para a investigação da aplicabilidade do Teste de Mutação para o teste de programas funcionais, escritos em SML. O Teste de Mutação é um critério de teste que fornece uma maneira de auxiliar na geração e na avaliação de um conjunto de casos de teste. Devido ao grande volume de informações que estão envolvidas na aplicação do Teste de Mutação, é essencial a existência de ferramentas de apoio para o uso desse critério. A fim de viabilizar a aplicação do Teste de Mutação para SML foi desenvolvida a ferramenta *web* PROTEUM/SML, que implementa os operadores de mutação definidos neste trabalho.

Palavras chaves: Teste de Mutação, Linguagem de Programação Funcional, Standard ML.

1 Introdução

Embora durante todo o processo de desenvolvimento de software sejam utilizadas técnicas, métodos e ferramentas a fim de evitar que erros sejam introduzidos no produto, a atividade de teste é de fundamental importância para a identificação e posterior eliminação dos erros que persistem, nas diversas fases do desenvolvimento. Além disso, a atividade de teste de software visa a fornecer evidências de confiabilidade e qualidade de um produto de software, em complemento a outras atividades, como por exemplo, o uso de revisões e de técnicas formais rigorosas de especificação e de verificação.

O Teste de Mutação, um critério de teste baseado em erros, procura revelar erros típicos cometidos no desenvolvimento de um produto. Um aspecto importante desse critério é fornecer uma maneira de auxiliar na geração e avaliação de um conjunto de casos de teste. Devido ao grande volume de informações que estão envolvidas na aplicação do Teste de Mutação, em que, geralmente, um grande número de produtos deve ser gerado, simulado e comparado, é essencial a existência de ferramentas de apoio para o uso desse critério. O desenvolvimento de ferramentas para o suporte à atividade de teste é de grande importância, uma vez que sua condução manual é propensa a erros, improdutiva e limitada a produtos muito simples.

Linguagens funcionais realizam computações por meio de definições e aplicações de funções [29]. Estas linguagens são livres de *side-effects*, não possuindo, portanto, comandos de atribuições e uma expressão sempre será avaliada com o mesmo valor. Segundo Claessen et al. [6], apesar de linguagens funcionais

apresentarem um estilo de programação que favorece o desenvolvimento de programas com menores taxas de erros, programas funcionais podem conter erros decorrentes da falta de entendimento de suas propriedades. Nesse contexto, é importante a investigação de critérios de teste para apoiar o teste de programas funcionais. Entretanto, existem poucas iniciativas para o teste de programas funcionais bem como no desenvolvimento de ferramentas que apóiem essa atividade [4, 13, 20, 21]. Além disso, tais iniciativas não possuem uma medida de cobertura da atividade de teste.

A linha de pesquisa deste trabalho é a investigação da adequabilidade da aplicação de critérios de teste tradicionalmente utilizados em programas imperativos no teste de programas funcionais. Em particular, busca-se estabelecer subsídios para a investigação da aplicação do Teste de Mutação para o teste de programas em SML (*Standard Meta Language*) [19]. A fim de viabilizar tal aplicação, foi desenvolvida a ferramenta PROTEUM/SML [31], que implementa os operadores de mutação para SML definidos por [30]. Os operadores de mutação caracterizam o Teste de Mutação para a linguagem/especificação alvo, estabelecendo os requisitos de teste a serem satisfeitos.

Este artigo está organizado da seguinte forma. Na seção 2 são apresentados os conceitos referentes ao Teste de Mutação e linguagens funcionais, enfatizando-se a linguagem SML. Na seção 2.2.1 são relacionados alguns trabalhos direcionados à VV&T (Verificação, Validação e Teste) de programas funcionais. Na seção 3 são descritos os operadores de mutação definidos para SML. Na seção 4 são apresentados as funcionalidades, a arquitetura e os aspectos de implementação relevantes da PROTEUM/SML. Na seção 5 é apresentado um estudo de caso para ilustrar a aplicação do Teste de Mutação para um programa SML. Finalmente, na seção 6 são feitas as considerações finais e sobre trabalhos futuros.

2 Conceitos Básicos

2.1 Teste de Mutação

O Teste de Mutação é um critério de teste baseado em erros, originalmente proposto para o teste de unidade [9]. Contudo, diversos pesquisadores têm aplicado seus conceitos fundamentais em vários contextos, tais como no teste de integração [8], teste de programas orientados a objeto [2, 14–16], teste de especificação [11, 12, 25, 28], teste de protocolo [22] e teste de modelo de segurança de rede [24]. Esse critério fornece ao testador um modo sistemático tanto para guiar a geração de casos de teste quanto para avaliar a qualidade do caso de teste. O Teste de Mutação utiliza produtos criados a partir do produto a ser testado (chamados de *mutantes*), com pequenas alterações sintáticas, que são modeladas por *operadores de mutação*. Em geral, os operadores estão associados a um tipo ou classe de erros que se pretende revelar no produto em teste.

O Teste de Mutação consiste em encontrar casos de teste que façam os mutantes comportarem-se diferentemente do produto original P, a fim de distinguir os mutantes. Os mutantes que foram distinguidos são ditos 'mortos'. Os mutantes 'vivos' são aqueles que se comportaram como o produto original para todo o conjunto T de casos de teste. Isso pode ocorrer i) porque o mutante é equivalente ao produto original ou ii) porque o conjunto de casos de teste não foi adequado o suficiente para distinguir o mutante. Na primeira situação, os mutantes podem ser desconsiderados. Na segunda situação, o conjunto T de casos de teste deve ser melhorado.

A adequação de um conjunto de casos de teste em relação a um programa em teste é obtida através do *escore de mutação*, que fornece a porcentagem de mutantes não-equivalentes mortos pelo conjunto de casos de teste, sendo definido da seguinte maneira:

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

sendo:
$DM(P,T)$: número de mutantes mortos por T;
$M(P)$: número total de mutantes gerados; e
$EM(P)$: número de mutantes gerados equivalentes a P.

O escore de mutação fornece ao testador um mecanismo de avaliar a qualidade da atividade de teste. Quando o escore de mutação alcança valor igual a 1.00, diz-se que o conjunto T de casos de teste é adequado em relação ao Teste de Mutação para testar o produto P, aumentando a confiança no produto em teste.

2.2 Linguagens Funcionais

Linguagens funcionais enfatizam a avaliação de expressões, ao invés da execução de comandos. As expressões são formadas usando-se funções, que são consideradas como valores tais como inteiro e *string*. Uma função pode retornar outra função, pode utilizar funções como parâmetros e, até mesmo, construir composição de

funções. Não há *side-effects* em programas funcionais, assim o valor da avaliação de uma expressão sempre será o mesmo [29].

SML (*Standard Meta Language*) caracteriza-se, principalmente, por ser uma linguagem de programação funcional. Essa linguagem foi proposta em 1983 e, posteriormente, definida formalmente em notações matemáticas por [18]. Uma revisão da linguagem, conhecida como SML'97, foi feita para torná-la mais simples, sendo descrita em [19].

Uma conseqüência da programação funcional é que a computação ocorre através das avaliações de expressões e não por atribuições a variáveis. Porém, SML difere das linguagens funcionais puras no sentido de permitir o uso de construtores tais como variáveis e operações de atribuição.

Um aspecto importante de SML é ser uma linguagem fortemente tipada, na qual todos os valores e variáveis possuem um tipo que pode ser determinado em tempo de compilação, sem a necessidade de executar o programa. Isso garante que apenas operações compatíveis sejam realizadas [23]. Esse processo de checagem de tipo permite que muitos erros sejam encontrados pelo compilador, não sendo necessária a execução do programa. Embora a maioria das linguagens fortemente tipadas requeira uma declaração de tipo de todas variáveis, em SML isso não é necessário e apenas se requer uma declaração de tipo quando é impossível de se deduzir o tipo [29].

Além disso, SML apóia polimorfismo que permite uma função ter argumentos de vários tipos. Outra característica de SML, visando ao desenvolvimento de programas de larga-escala, é de fornecer facilidades como de modularidade, encapsulamento de conceitos e reuso de software, através de *structures* (conjunto de tipos de dados, funções, exceções e demais elementos), *signatures* (tipo para *structure*) e *functors* (função que possui como argumento *structures* e outros elementos, e retorna um *structure*) [29].

SML fornece um mecanismo de tratamento de exceções, sendo que as exceções podem incluir valores arbitrários, inclusive funções. Em qualquer ponto do código, uma exceção pode ocorrer, abortando a operação atual e retornando o controle para o último tratador definido para tal exceção [23].

2.2.1 Programas Funcionais: VV&T

A investigação de critérios de teste para apoiar o teste de programas funcionais é de grande importância para a revelação de erros decorrentes da falta de entendimento de suas propriedades. Entretanto, existem poucas iniciativas para o teste de programas funcionais bem como no desenvolvimento de ferramentas que dêem apoio a essa atividade.

Uma ferramenta que pode-se citar é a *QuickCheck* [4, 5] que apóia o teste de programas Haskell, uma linguagem de programação funcional pura. Tal ferramenta testa propriedades dos programas, as quais são descritas como funções Haskell. O teste pode ser feito automaticamente por entradas aleatórias ou pelo testador. Porém, com essa abordagem de teste aleatório, *QuickCheck* torna-se limitado a pequenos programas e não possui uma medida de cobertura que contribui para avaliar a qualidade e a adequação da atividade de teste.

HUnit [13] é um *framework* de teste de unidade para Haskell, similar à ferramenta JUnit para Java. Permite que casos de teste sejam estruturados hierarquicamente, podendo ser executados automaticamente.

Auburn [20] é uma ferramenta para *benchmarking* de estruturas de dados funcionais. Produz uma árvore que classifica as melhores estruturas de dados de acordo como é utilizado. Possui a capacidade de extrair um perfil de como uma aplicação usa uma estrutura de dados.

CaslTest [21] é uma ferramenta que auxilia no teste de especificações CASL utilizando a linguagem SML. CASL é uma linguagem unificada para descrever especificações algébricas. O objetivo da ferramenta é extrair casos de teste das especificações CASL e projetar e gerar os oráculos de teste correspondentes e dados de teste em SML. Dessa maneira, verificam-se se os casos de teste derivados dos axiomas da especificação são satisfeitos pelos programas escritos em SML. Entretanto, CaslTest também não estabelece nenhuma medida de cobertura do teste realizado.

Devido à ausência de critérios de teste que possuam medida de cobertura da atividade de teste em programas funcionais, neste trabalho busca-se fornecer subsídios para a investigação da aplicação do Teste de Mutação em programas SML. O Teste de Mutação, através do escore de mutação, permite obter a adequação de um conjunto de casos de teste em relação a um programa em teste. Na próxima seção são descritos os operadores de mutação para SML definidos por [30], os quais são implementados na ferramenta Proteum/SML, que oferece apoio ao Teste de Mutação para programas SML.

3 Operadores de Mutação para SML

A definição dos operadores de mutação compreende um dos aspectos mais importantes para a aplicação do Teste de Mutação. Os operadores caracterizam o critério, estabelecendo os requisitos de teste a serem satisfeitos. Em geral, o desenvolvimento do conjunto de operadores está relacionado com uma classe de

erros típicos que podem ser cometidos no contexto de uma determinada linguagem/especificação. Assim, os operadores de mutação devem representar a "implementação" dessa classe de erros.

Além dos tipos de erros que se desejam revelar e da cobertura que se quer garantir, a escolha de um conjunto de operadores de mutação depende também da linguagem em que os programas estão escritos. Por exemplo, em [3] encontra-se a relação de 22 operadores de mutação utilizados por um sistema de mutação para programas em Fortran. Para a linguagem C, Agrawal [1] define um conjunto de 71 operadores de mutação, os quais foram implementados na ferramenta Proteum/C [7].

Nesta seção é apresentado o conjunto de 17 operadores de mutação definido para a aplicação do Teste de Mutação para a linguagem SML e implementado na PROTEUM/SML. Os operadores são classificados em função do elemento no qual é realizado a mutação. A classificação e a quantidade de operadores definidos para cada classe são dadas a seguir: expressões (3), operadores (5), constantes (4) e variáveis (5). O domínio de um operador de mutação é estabelecido em termos da entidade sintática que é afetada. Os nomes dos operadores são formados por três letras maiúsculas, cuja primeira letra indica a classe do operador: **E** (*expression*), **O** (*operator*), **C** (*constant*) e **V** (*variable*), e as demais letras indicam a descrição do operador. Por exemplo, o operador VDT pertence a classe de variáveis e verifica a cobertura do domínio das variáveis (*Domain Trap*). Na Tabela 1 é apresentado resumidamente esse conjunto de operadores de mutação.

Tabela 1: Operadores de Mutação definidos na PROTEUM/SML

Operador	*Descrição*
EDL	*Expression **D**e**L**etion*
	Exclui expressões mas mantém o ponto-e-vírgula no final da expressão
EME	***M**atch **E**xpression Mutation*
	Adiciona a função `trap_on_match()` em cada *pattern* de *matches*
ETI	***T**rap on **I**f Condition*
	Substitui a condição c da expressão `if-then-else` pelas funções `trap_on_true(c)` e `trap_on_false(c)`
OAR	***A**rithmetic Operator **R**eplacement*
	Substitui cada operador aritmético (`+, -, *, /,div,mod`) por outro operador aritmético
ORR	***R**elational Operator **R**eplacement*
	Substitui cada operador relacional ($=, <, >, <=, >=, <>$) por outro operador relacional
OLR	***L**ogical Operator **R**eplacement*
	Substitui cada operador lógico (`andalso, orelse`) por outro operador lógico
OLN	***L**ogical **N**egation*
	Nega as expressões lógicas com `not`
OLC	***L**ogical **C**ontext Negation*
	Nega as condições da expressão `if-then-else` e `while-do` com `not`
CBR	***B**oolean **R**eplacement*
	Substitui a constante `true` por `false` e vice-versa
CCC	***C**onstant for **C**onstant Replacement*
	Troca cada constante (global/local) do programa por outras constantes (global/local)
CCS	***C**onstant for **S**calar Replacement*
	Troca cada variável (global/local) do programa por todas as constantes (global/local)
CRC	***R**equired **C**onstant Replacement*
	Troca cada variável por constantes requeridas
VAV	***A**nonymous **V**ariable*
	Substitui cada variável escalar (global/local) pela variável anônima
VDT	***D**omain **T**raps*
	Substitui cada variável pelas funções `trap_on_negative_integer(`x`)`, `trap_on_positive_integer(`x`)`, `trap_on_zero_integer(`x`)`, `trap_on_negative_real(`x`)`, `trap_on_positive_real(`x`)` `trap_on_zero_real(`x`)` `trap_on_true(`x`)`, `trap_on_false(`x`)`
VRF	***R**ecord **F**ield Mutation*
	Substitui o *label* do operador # por outros *labels* de registros
VSV	***S**calar **V**ariable Reference Replacement*
	Substitui cada variável escalar (global/local) por outra (global/local)
VTM	***T**widdle **M**utations*
	Substitui cada referência escalar x pelo predecessor imediato e o sucessor imediato do valor corrente do argumento, através das funções: `pred_integer(`x`)`, `succ_integer(`x`)`, `pred_real(`x`)` e `succ_real(`x`)`

Na Figura 1 são apresentados um programa em SML e um mutante gerado pela aplicação do operador de mutação ORR. Esse programa retorna verdadeiro se uma lista de caracteres é vazia ou é formada por letras ou números, caso contrário retorna falso. O operador ORR substitui cada operador relacional ($=, <, >, <=, >=, <>$) por outro operador relacional, a fim de verificar se a escolha do operador relacional está correta dentro de uma expressão.

Embora a linguagem SML apresente características de encapsulamento, polimorfismo e modularidade

```
fun valid_f nil = true
  | valid_f (y::ys) =
      if (((y >= #"A") andalso (y <= #"Z")) orelse
         ((y >= #"a") andalso (y <= #"z")) orelse
         ((y >= #"0") andalso (y <= #"9")))
      then valid_f(ys)
      else false;
```

Programa Original

```
fun valid_f nil = true
  | valid_f (y::ys) =
      if (((y > #"A") andalso (y <= #"Z")) orelse
         ((y >= #"a") andalso (y <= #"z")) orelse
         ((y >= #"0") andalso (y <= #"9")))
      then valid_f(ys)
      else false;
```

Mutante

Figura 1: Exemplo de Programa Mutante

através de *structures*, *signatures* e *functors*, não foram propostos operadores de mutação para tais aspectos. O conjunto de operadores de mutação definido por [30] é direcionado apenas para o teste de unidade, não sendo abrangente para o teste de integração.

A aplicação do conjunto de operadores de mutação em um programa P gera um conjunto $\phi(P)$ de mutantes. Um conjunto T de casos de teste é adequado a P com relação a $\phi(P)$, se para cada programa $M \in \phi(P)$, ou M é equivalente a P, e nesse caso M e P possuem o mesmo comportamento para todo domínio de entrada, ou M difere de P em no mínimo um ponto de teste. Para distinguir o comportamento do mutante M do programa original P, analisam-se suas saídas após a execução com o conjunto T de casos de teste. Considere uma expressão e em um programa P e e_m a mesma expressão mas contendo alguma mutação definida por um operador de mutação *op* que gera o mutante M. Em geral, um caso de teste t precisa satisfazer três condições para distinguir M e P [10]:

- Alcançabilidade: e_m precisa ser executado.
- Necessidade: O estado de M imediatamente após alguma execução de e_m precisa ser diferente do estado de P imediatamente após a execução de e.
- Suficiência: A diferença nos estados de P e M imediatamente após a execução de e_m e e precisa ser propagada até o final da execução de P e M tal que os estados finais alcançados entre eles quando executado com T sejam diferentes.

4 Proteum/SML

Proteum/SML é uma ferramenta *web* desenvolvida para dar apoio ao Teste de Mutação para programas em SML que implementa o conjunto de 17 operadores de mutação apresentado na Tabela 1.

4.1 Aspectos Operacionais

A Proteum/SML oferece um conjunto mínimo de operações que satisfaz os requisitos básicos de uma ferramenta de teste baseada em mutação:

- Tratamento de casos de teste: execução, inclusão/exclusão e habilitação/desabilitação;
- Tratamento de mutantes: geração, seleção, execução e análise; e
- Análise de adequação: escore de mutação e relatórios estatísticos.

A organização da Proteum/SML é feita por projetos e sessões, a qual permite minimizar o número de gerações de mutantes e execuções de casos de teste. Isso é de grande importância para a condução de estudos empíricos que utilizam um produto a ser testado com o mesmo conjunto de mutantes (ou um subconjunto desse) e/ou com casos de teste similares.

Cada projeto determina o código fonte do programa a ser testado e o conjunto de operadores que serão utilizados para a geração dos mutantes. Um projeto pode possuir uma ou mais sessões de teste. Uma sessão pode importar qualquer subconjunto dos mutantes gerados no projeto a qual pertence, informando a porcentagem ou quantidade de mutantes de cada operador ou, então, selecionando o número do mutante ou as linhas que se deseja realizar a mutação. Na Figura 2 é apresentada a página de criação de uma sessão

de teste, na qual são mostrados o menu da ferramenta (área 1), o formulário para a criação de uma sessão (área 2), uma tabela que resume as informações do projeto corrente (área 3) e uma tabela que lista as sessões do projeto corrente (área 4).

Existe também a noção de grupo de sessões que facilita o trabalho cooperativo entre um grupo de testadores. Um grupo é formado por usuários de várias sessões de um mesmo projeto, sendo que os mutantes são particionados entre as sessões do grupo conforme o número máximo de mutantes que cada um pode possuir.

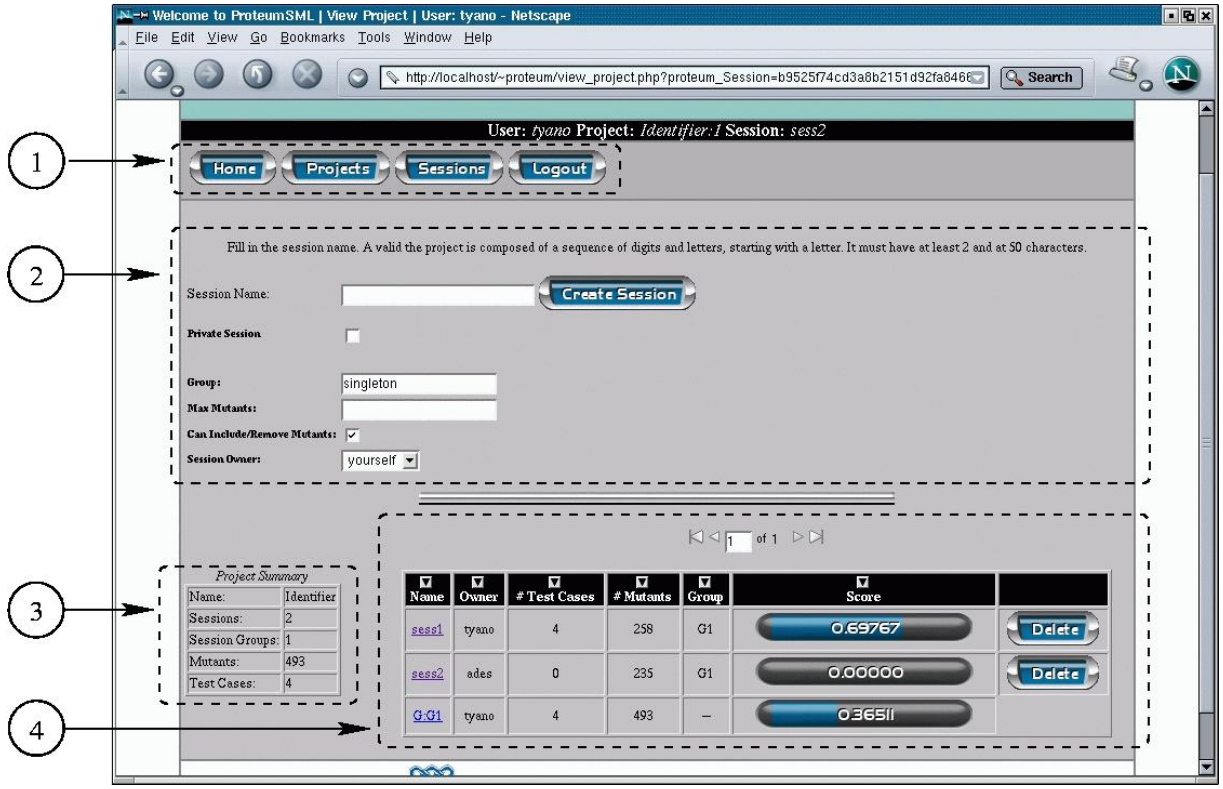

Figura 2: Página de Criação de Sessão

O acesso à ferramenta é controlado através de contas de usuário protegidas por senha. Existem três categorias de usuário: administrador, proprietário de projeto e usuário de sessão. O usuário administrador é criado na instalação da PROTEUM/SML e é responsável pelas autorizações à entrada de outros usuários ao sistema. Um proprietário de projeto tem permissão para criar novos projetos e determina quais usuários podem pertencer ao seu projeto. Já o usuário de sessão tem permissão para criar sessões em um projeto já existente.

Após a criação de uma sessão, pode-se incluir casos de teste tanto interativamente quanto por importação. Em ambos os modos de inclusão, a saída é mostrada ao usuário que deve verificar se está correta, confirmando ou não a inclusão do caso de teste. Na inclusão interativa, o usuário fornece o caso de teste no campo do formulário respectivo. Na importação, o usuário fornece um arquivo ASCII com o caso de teste.

Após a execução dos mutantes com o conjunto de caso de teste, aqueles que permaneceram vivos devem ser analisados para verificar se são equivalentes ou não ao programa em teste. A análise pode ser feita observando-se as linhas do código nas quais a mutação ocorreu. Para tanto, a ferramenta permite a visualização lado a lado do programa em teste e de um mutante, destacando-se as linhas mutadas.

A ferramenta disponibiliza ao testador dois tipos de relatórios estatístico: por operador e por caso de teste. No primeiro, é apresentada a quantidade de mutantes vivos, mortos e equivalentes referentes a cada operador. No segundo, resumem-se informações sobre a execução de cada caso de teste. Tais relatórios permitem realizar uma análise detalhada da adequação dos casos de teste em relação aos mutantes gerados.

Na Figura 3 é apresentada a página que lista os mutantes de uma sessão de teste. Na área 1 encontram-se informações sobre o *status* da sessão, verificando-se principalmente o escore de mutação da sessão que indica a adequação dos casos de teste. Na área 2 podem-se observar os mutantes gerados na sessão.

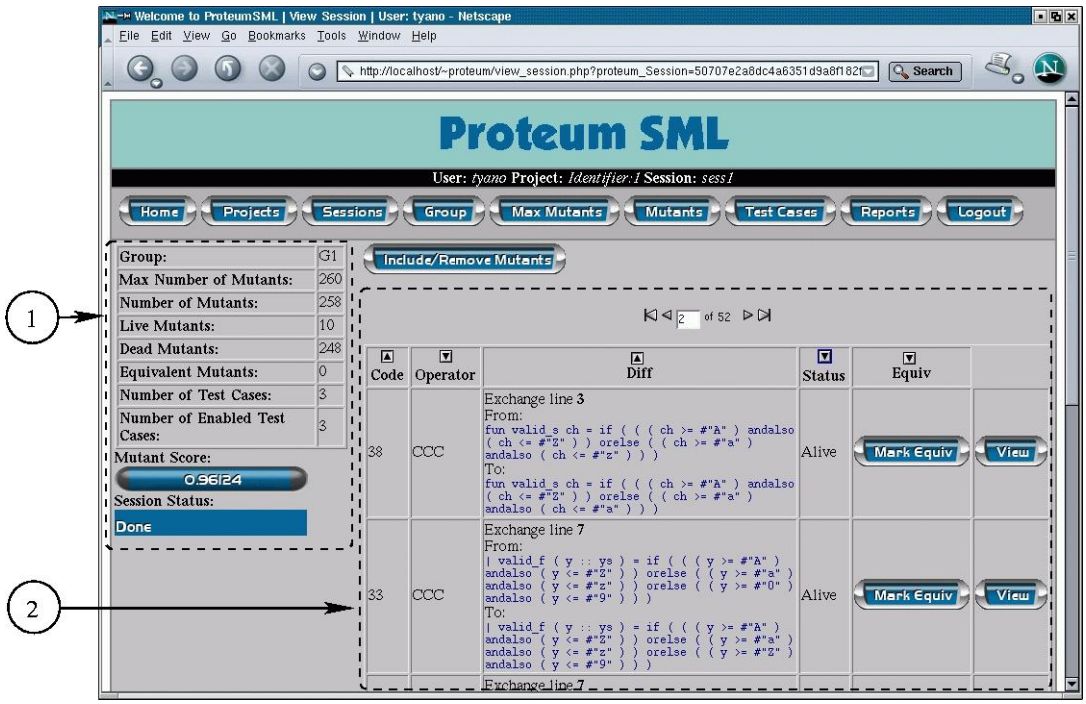

Figura 3: Mutantes de uma Sessão

4.2 Arquitetura e Aspectos Relevantes de Implementação

A arquitetura da PROTEUM/SML é apresentada na Figura 4. As funcionalidades da ferramenta encontram-se divididas entre módulos independentes e dependentes do Teste de Mutação para SML. Os módulos independentes são compostos pelas funcionalidades comuns entre a PROTEUM/SML e a ferramenta PROTEUM/CPN, que está sendo desenvolvida para dar apoio ao Teste de Mutação para Redes de Petri Coloridas [26]. Buscou-se, dessa forma, identificar e isolar as funcionalidades de uma ferramenta de apoio ao Teste de Mutação que sejam independentes da linguagem/especificação alvo, promovendo o reuso de código. Por sua vez, os módulos dependentes possuem as funcionalidades que são estritamente relacionadas à SML. Por exemplo, é necessário realizar um tratamento no código fonte de programas SML para sua execução.

A interação da ferramenta é realizada através de interface *web*. O módulo gerenciador de *skins* é responsável pela formatação das páginas, separando os demais módulos de detalhes específicos de exibição. Dessa forma, pode-se alterar toda a aparência das páginas que compõem a ferramenta sem a necessidade de modificar os códigos referentes às funcionalidades. Além disso, é possível personalizar a interface para se adequar às necessidades de uma linguagem/especificação específica.

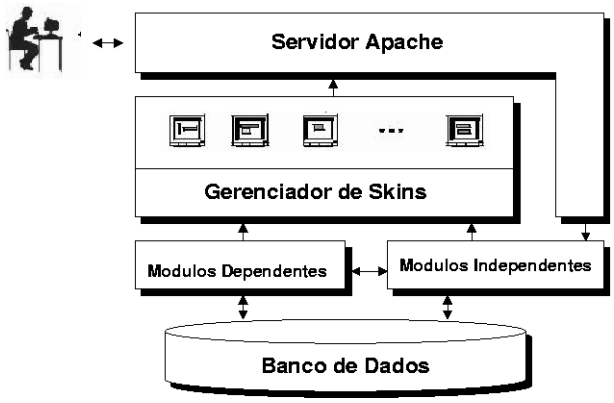

Figura 4: Arquitetura da PROTEUM/SML

Os módulos da PROTEUM/SML foram desenvolvidos na linguagem *PHP 4.1.2*, sobre a plataforma *Linux RedHat 7.3*, kernel versão 2.4.20-18.7, utilizando o *Apache 1.1* como servidor de *WWW* e o *MySQL 3.23.56* como servidor de banco de dados.

Os operadores de mutação, utilizados para a geração dos mutantes na PROTEUM/SML, foram descritos na linguagem $\mathcal{MuDeL}$ (*MUtant DEscription Language*) [27]. A linguagem $\mathcal{MuDeL}$ visa a fornecer precisão e formalidade na descrição dos operadores e a facilitar a automatização da geração de mutantes. Considerando o aspecto crucial dos operadores de mutação no Teste de Mutação, isso é grande importância para se evitar ambigüidades e inconsistências. Foi utilizado o sistema `mudelgen` para "compilar" a descrição dos operadores em $\mathcal{MuDeL}$ e gerar os mutantes, com base na gramática livre de contexto de SML.

O presente trabalho adotou SML'97 e o compilador SML/NJ (*Standard ML of New Jersey*) desenvolvido pelos Laboratórios Bell e pela Universidade de Princeton.

5 Estudo de Caso

Nesta seção é apresentado um estudo de caso da aplicação do Teste de Mutação para SML com o programa exemplo *identifier.sml* (Figura 5), utilizando-se a ferramenta PROTEUM/SML. Esse programa é a implementação em SML do programa *identifier* escrito na linguagem C apresentado em [17]. O programa a ser testado é um verificador de identificadores, que analisa se um determinado identificador é válido. Para ser válido, o identificador deve ser uma cadeia de letras e dígitos, iniciada por uma letra e ter no mínimo 1 e no máximo 6 caracteres.

```
fun no_new_line (#"n"::nil) = nil
| no_new_line (z::zs) = z::no_new_line(zs);

fun valid_s ch =
     if (((ch >= #"A") andalso (ch <= #"Z")) orelse
     ((ch >= #"a") andalso (ch <= #"z")))
     then true
     else false;

fun valid_f nil = true
| valid_f (y::ys) =
     if (((y >= #"A") andalso (y <= #"Z")) orelse
     ((y >= #"a") andalso (y <= #"z")) orelse
     ((y >= #"0") andalso (y <= #"9")))
     then valid_f(ys)
     else false;

fun valid_all nil = false
| valid_all (x::xs) =
     if valid_s(x)
     then valid_f(xs)
     else false;

fun ident s =
     let
          val value_l = no_new_line(explode(s));
          val length_l = length(value_l)
     in
          if (valid_all(value_l) andalso length_l >= 1 andalso length_l <= 6)
          then "Valid\n"
          else "Invalid\n"
     end;

print("Input a identifier\n");
val value = TextIO.inputLine(infile);
val result = ident(value);
print(result);
```

Figura 5: Programa *identifier.sml*

Na Tabela 2 são apresentadas algumas informações sobre o programa *identifier.sml*, tais como o número de linhas de código, de variáveis, de constantes, de operadores e de expressões, as quais influenciam na quantidade de mutantes gerados no Teste de Mutação desse programa.

Tabela 2: Informações do Programa *identifier.sml*

Descrição	*Qtde*
Linhas de Código	25
Variáveis	12
Constantes	12
Operadores	21
Expressões	17

Geração de Mutantes

A quantidade de mutantes gerados por cada operador na PROTEUM/SML para o programa exemplo é apresentada na Tabela 3. A aplicação de todos os operadores gera 493 mutantes, sendo 248, 104, 10, 131 pelos operadores de mutação de constantes, operadores, expressões e variáveis, respectivamente.

Tabela 3: Mutantes do Programa *identifier.sml*

Operador	*Qtde*	*Operador*	*Qtde*
CBR	6	EDL	2
CCC	80	EME	0
CCS	115	ETI	8
CRC	47	VAV	4
OAR	0	VDT	0
OLC	4	VRF	0
OLN	30	VSV	127
OLR	10	VTM	0
ORR	60		
Quantidade Total: 493			

Criação de Casos de Teste

Foi construído um conjunto T de casos de teste adequado aos mutantes do programa *identifier.sml* gerados por todos os operadores de mutação da PROTEUM/SML. Um caso de teste é denotado pelo par de entrada e e a saída s: (e, s).

Inicialmente, foi inserido o conjunto de casos de teste $T_0 = \{(a1, \textit{válido}), (2B3, \textit{inválido}), (Z - 12, \textit{inválido}), (A1b2C3d, \textit{inválido})\}$. Após a execução dos mutantes com T_0, o escore de mutação obtido é 0.77282. A fim de melhorar o escore de mutação foi adicionado o conjunto de casos de teste $T_1 = \{(zzz, \textit{válido}), (aA, \textit{válido}), (A1234, \textit{válido}), (aa09, \textit{válido})\}$, obtendo escore igual a 0.88641. Mesmo com a inclusão de T_0 e T_1, o escore de mutação ainda não é satisfatório. Assim, foi elaborado o conjunto de casos de teste $T_2 = \{(1\#, \textit{inválido}), (AAA, \textit{válido}), ([, \textit{inválido}), (\{, \textit{inválido}), (x :, \textit{inválido}), (x[[, \textit{inválido}), (x\{\{, \textit{inválido}), (@, \textit{inválido}), (xy\_, \textit{inválido})\}$, melhorando o escore para 0.97363. A partir de então, foram identificados quatro mutantes equivalentes. Esses mutantes permaneceram vivos mesmo após a inclusão de T_2 e, ao analisá-los, foi observado que eles teriam sempre o mesmo comportamento do programa original, assim nenhum caso de teste poderia matá-los. Ao incluir o conjunto de caso de teste $T_3 = \{(c, \textit{válido}), (ZZZZ, \textit{válido})\}$ e $T_4 = \{(ABCEDF, \textit{válido})\}$ foi obtido escore de mutação igual a um e o conjunto de casos de teste $T = T_0 \cup T_1 \cup T_2 \cup T_3 \cup T_4$ constitui um conjunto 100% adequado ao Teste de Mutaçao para o programa *identifier.sml*.

Na Tabela 4 mostra-se a evolução da atividade de teste no programa *identifier.sml* com os conjuntos de casos de teste que compõem T. São mostrados os valores do escore de mutação a cada inclusão de T_i.

Tabela 4: Evolução do Escore de Mutação do Programa *identifier.sml*

Conj. de casos de teste	*Escore de mutação*
T_0	0.77282
$T_0 \cup T_1$	0.88641
$T_0 \cup T_1 \cup T_2$	0.97363
$T_0 \cup T_1 \cup T_2 \cup T_3$	0.98986
$T_0 \cup T_1 \cup T_2 \cup T_3 \cup T_4$	1.0000

6 Conclusão

Atualmente, há poucos trabalhos relacionados ao teste de programas funcionais, sendo que os existentes não possuem uma medida de cobertura da atividade de teste. Assim, neste trabalho, buscou-se estabelecer subsídios para a investigação da adequabilidade da aplicação de critérios de teste tradicionalmente utilizados em programas imperativos no teste de programas funcionais. Particularmente, investigou-se a aplicabilidade do critério Teste de Mutação para o teste de programas em SML (*Standard Meta Language*). O Teste de Mutação é um critério de teste que fornece uma maneira de auxiliar na geração e na avaliação de um conjunto de casos de teste, sendo explorado em diversos contextos, como no teste de unidade, de integração e de especificação.

Um dos principais aspectos para caracterizar o Teste de Mutação é a definição de um conjunto de operadores de mutação específico para a linguagem alvo. A escolha dos operadores determina tanto os tipos de erros que se deseja revelar e a cobertura que se quer garantir na atividade de teste, como também determina a eficiência da utilização do Teste de Mutação, uma vez que o custo computacional está relacionado ao número de mutantes gerados pela aplicação dos operadores de mutação. Neste trabalho, foi apresentado um conjunto de 17 operadores de mutação para a linguagem SML definido em [30]. Os operadores de mutação foram descritos na linguagem $\mathcal{MuDeL}$, a qual fornece precisão e formalidade na descrição dos mesmos como também facilita a automatização da geração de mutantes.

A disponibilidade de ferramentas de apoio ao Teste de Mutação contribui para aumentar a produtividade, visto que a quantidade de informações que deve ser tratada é muito grande. Para dar apoio à aplicação do Teste de Mutação para SML, foi desenvolvida a ferramenta *web* PROTEUM/SML. Sua implementação teve como base a ferramenta PROTEUM/CPN [26], a qual oferece apoio ao Teste de Mutação de Redes de Petri Coloridas.

Com o objetivo de avaliar a eficácia dos operadores de mutação definidos para SML, deve-se realizar estudos empíricos com exemplos que abrangem as principais características da linguagem. A fim de complementar o conjunto de operadores de mutação para SML, pode-se investigar operadores que explorem as características de encapsulamento, polimorfismo e modularidade de SML. Além disso, como foram definidos apenas operadores para o teste de unidade, pode-se investigar operadores para o teste de integração.

Outro trabalho que pode ser realizado é aplicar o Teste de Mutação de SML no código da especificação em Redes de Petri Coloridas produzida pela ferramenta PROTEUM/CPN, que também é escrito em SML. O objetivo seria a investigação da relação entre o teste de especificações de sistemas e das respectivas implementações desses sistemas, com enfâse no Teste de Mutação, utilizando-se as ferramentas PROTEUM/SML e PROTEUM/CPN. Na essência o propósito desse trabalho, seria buscar o relacionamento entre os conceitos de teste aplicados no nível de especificação e os do nível de implementação. No caso particular do Teste de Mutação, buscar-se-ia o relacionamento entre os operadores de mutação em nível de especificação e os operadores de mutação em nível de implementação.

Além disso, com a implementação da PROTEUM/SML viabiliza-se a realização de estudos empíricos que visem à investigação do relacionamento dos operadores de mutação definidos para SML com os de C. O propósito desse estudo seria buscar a relação dos conceitos de teste aplicados na programação funcional e imperativa.

Com base na experiência no desenvolvimento da PROTEUM/SML, juntamente com da ferramenta PROTEUM/CPN, motiva-se a abstração de um *framework* para o desenvolvimento de ferramentas de apoio ao Teste de Mutação, visto que foram identificadas as funcionalidades indenpendentes à linguagem/especificação alvo desse critério.

Referências

[1] Agrawal, H. (1989). Design of mutant operators for the C programming language. Relatório Técnico SERC-TR-41-P, Software Engineering Research Center/Purdue University.

[2] Bieman, J. M., Ghosh, S., e Alexander, R. T. (2001). A technique for mutation of Java objects. In *16th IEEE Internation Conference on Automated Software Engineering*, pp. 23–26, San Diego, CA.

[3] Budd, A. T. (1981). *Mutation Analysis: Ideas, Examples, Problems and Prospects*, cap. Computer Program Testing, pp. 129–148. North-Holland Publishing Company.

[4] Claessen, K. e Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In *Fifth ACM SIGPLAN international conference on Functional programming*, pp. 268–279. ACM Press.

[5] Claessen, K. e Hughes, J. (2002). Testing monadic code with quickcheck. *SIGPLAN Not.*, 37(12):47–59.

[6] Claessen, K., Runciman, C., Chitil, O., Hughes, J., e Wallace, M. (2002). Testing and tracing lazy functional programs. In 4^{th} *Summer School in Advanced Functional Programming*, Oxford.

[7] Delamaro, M. E. (1993). Proteum – Um ambiente de teste baseado na análise de mutantes. Dissertação de mestrado, ICMC/USP, São Carlos, SP.

[8] Delamaro, M. E., Maldonado, J. C., e Mathur, A. P. (2001). Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

[9] DeMillo, R. A., Lipton, R. J., e Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

[10] DeMillo, R. A. e Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910.

[11] Fabbri, S. C. P. F., Maldonado, J. C., Delamaro, M. E., e Masiero, P. C. (1999a). Proteum/FSM: A tool to support finite state machine validation based on mutation testing. In *XIX SCCC - International Conference of the Chilean Computer Science Society*, pp. 96–104, Talca, Chile.

[12] Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., e Masiero, P. C. (1999b). Mutation testing applied to validate specifications based on statecharts. In *10^{th} International Symposium on Software Reliability Engineering (ISSRE'99)*, pp. 210–219, Boca Raton, Flórida, EUA.

[13] Herington, D. (2002). *HUnit 1.0 User's Guide*.

[14] Kim, S., Clark, J. A., e Mcdermid, J. A. (2000). Class mutation: Mutation testing for object-oriented programs. In *Object-Oriented Software Systems – OOSS*.

[15] Kim, S., Clark, J. A., e Mcdermid, J. A. (2001). Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11(4).

[16] Ma, Y.-S., Kwon, Y.-R., e Offutt, J. (2002). Inter-class mutation operators for Java. In *13th International Symposium on Software Reliability Engineering - ISSRE'2002*, pp. 352–366, Annapolis, MD. IEEE Computer Society Press.

[17] Maldonado, J. C., Barbosa, E. F., Vincenzi, A. M. R., Delamaro, M. E., Souza, S. R. S., e Jino, M. (2003). Introdução ao teste de software. In *XVII SBES – Simpósio Brasileiro de Engenharia de Software (Minicurso)*, Manaus, AM.

[18] Milner, R., Tofte, M., e Harper, R. (1990). *The Definition of Standard ML*. The MIT Press, Cambridge, Mass.

[19] Milner, R., Tofte, M., Harper, R., e MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass.

[20] Moss, G. E. e Runciman, C. (1999). Automated benchmarking of functional data structures. *Lecture Notes in Computer Science*, 1551:1–15.

[21] Oliveira, K. A., Machado, P. D. L., e Andrade, W. L. (2003). CaslTest - test case, test oracle and test data generation from Casl specifications. In *Sessão de Ferramentas do XVII SBES – Simpósio Brasileiro de Engenharia de Software*, pp. 73–78, Manaus, AM.

[22] Probert, R. L. e Guo, F. (1991). Mutation testing of protocols: Principles and preliminary experimental results. In *IFIP TC6 Third International Workshop on Protocol Test Systems*, pp. 57–76, North-Holland.

[23] Pucella, R. (2001). Notes on programming Standard ML of New Jersey. Relatório Técnico Version 110.0.6, Departament of Computer Science, Cornell University, Ithaca, NY.

[24] Ritchey, R. W. (2000). Mutation network models to generate network security test cases. In *Mutation 2000 Symposium*, pp. 101–108, San Jose, California.

[25] Simão, A. S., Maldonado, J. C., e Fabbri, S. C. P. F. (2000). Proteum-RS/PN: A tool to support edition, simulation and validation of Petri nets based on mutation testing. In *XIV SBES – Simpósio Brasileiro de Engenharia de Software*, pp. 227–242, João Pessoa, PB.

[26] Simão, A. S. (2002). *Teste e Validação de Redes de Petri Coloridas Usando Análise de Mutantes*. Qualificação de doutorado, ICMC/USP, São Carlos, SP.

[27] Simão, A. S. e Maldonado, J. C. (2002). MuDeL: A language and a system for describing and generating mutants. *Journal of the Brazilian Computer Society*, 8(1):73–86.

[28] Souza, S. R. S., Maldonado, J. C., Fabbri, S. C. P. F., e Lopes de Souza, W. (2000). Mutation testing applied to Estelle specifications. In *33^{rd} Hawaii Internacional Conference on System Sciences, Mini-Tracks: Distributed Systems Testing*, Maui, Havaii.

[29] Ullman, J. D. (1998). *Elements of ML Programming - ML97 Edition*. Prentice Hall, New Jersey, USA.

[30] Yano, T. (2004). Estudo do teste de mutação em programas funcionais sml. Dissertação de mestrado, ICMC/USP, São Carlos, SP.

[31] Yano, T., Simao, A. S., e Maldonado, J. C. (2003). Proteum/SML: Uma ferramenta de apoio ao teste de mutação para a linguagem Standard ML. In *Sessão de Ferramentas do XVII SBES – Simpósio Brasileiro de Engenharia de Software*, pp. 67–72, Manaus, AM, Brasil.