3
Modelos Gramaticais

Inspirado no paradigma de teste baseado em modelo (Utting et al., 2006), este trabalho adota ***gramáticas*** como modelo de teste funcional, que pode ser usado ao longo de todo o ciclo de vida de um sistema (Ince, 1987). Esta dissertação propõe uma nova abordagem que consiste em construir gramáticas capazes de capturar todos os aspectos relevantes ao teste funcional de um sistema e representar computacionalmente estas gramáticas como modelos orientados a objetos. Os modelos provenientes destas gramáticas são denominados ***modelos gramaticais***.

Segundo esta proposta, cada gramática aborda um único aspecto de teste e todas são construídas respeitando as regras de uma ***gramática de referência***, conhecida como ***metagramática de teste***. A metagramática de teste define a linguagem de formação de regras das gramáticas de teste e quais são os elementos disponíveis a sua constituição. A cada gramática está associado um modelo orientado a objeto que interpreta as regras e os elementos da gramática de acordo com o aspecto de teste por ela retratado.

Os aspectos de interesse ao teste funcional de um sistema são os ***conceitos***, as ***funcionalidades***, o ***ambiente*** e os ***casos de teste*** usados na sua verificação. Como o enfoque dado a esta pesquisa é o desenvolvimento de testes, em paralelo ou não ao desenvolvimento do software, a captura destes aspectos por gramáticas usualmente se restringe aos conceitos e funcionalidades que pertencem ao escopo de verificação do sistema em teste (seção 2.1). Os casos de teste do sistema e suas variações de acordo com a técnica de teste funcional escolhida (seção 2.4.1) abordam apenas estes conceitos e funcionalidades. O ambiente de teste, seu estado inicial e suas configurações, dependentes ou não de plataforma, também são retratados neste mesmo contexto.

Este capítulo apresenta a definição de uma metagramática de teste (seção 3.1), descreve sua sintaxe (seção 3.2) e indica as possíveis variações semânticas adotadas no teste funcional de um sistema (seção 3.3) segundo esta pesquisa.

Destaca os trabalhos relacionados na literatura de teste de software que influenciaram a escolha deste tipo de representação e a opção por um paradigma de teste baseado em modelos (seção 3.4). Por fim, o capítulo detalha os modelos de teste criados a partir de gramáticas, estabelecendo o relacionamento entre a gramática e os conceitos de programação orientada a objetos utilizados pelo teste funcional (seção 3.5) retratado a seguir por este trabalho (capítulo 4).

3.1. Metagramática e Gramáticas de Teste

A ***metagramática de teste*** é uma gramática como qualquer outra. O que a diferencia das demais é o seu uso. A metagramática de teste é a gramática de referência que define a linguagem usada para construir as demais gramáticas aplicadas ao teste funcional de um sistema.

As ***gramáticas de teste***, descritas segundo a linguagem definida pela metagramática de teste, são gramáticas que representam os requisitos apontados pela especificação de um sistema em teste, os requisitos do seu teste e os requisitos do seu ambiente de operação. Ao invés de serem utilizadas como reconhecedores de representações, elas são utilizadas como geradores de modelos. No contexto de teste funcional, elas são aplicadas, por exemplo, na geração dos dados de um caso de teste.

Uma ***gramática*** descreve um conjunto de padrões em termos de um léxico finito e de um conjunto finito de regras que especificam as combinações permitidas de elementos neste léxico (Arsanjani, 2005).

O ***léxico*** define o ***vocabulário***, possivelmente categorizado em classes de lexemas (ex. nomes de variáveis, nomes de funções, nomes de constantes, nomes de tipos ou classes, nomes de operadores, etc.). O conjunto de ***regras***, constituídas pelos elementos do léxico, define a ***sintaxe*** da gramática. O significado do léxico e das regras, sua ***semântica***, varia de acordo com a interpretação da gramática. Cada interpretação corresponde, em termos computacionais, às transformações (ex. geração de código) que podem ser realizadas com a sua estrutura. Esta interpretação é feita gradativamente em níveis: análise léxica, análise sintática e análise semântica.

A análise léxica decompõe um texto em seus elementos. A análise sintática estabelece os relacionamentos entre os elementos, que definem a sua estrutura. A análise semântica revela o que a estrutura significa.

3.2. Sintaxe das Gramáticas de Teste

Usualmente gramáticas são definidas como um texto seqüencial similar à descrição de um programa. Gramáticas também podem ser representadas como máquinas de estado que é tipicamente o caso para analisadores léxicos. Nesta pesquisa é empregado o formato de texto para descrever tanto a metagramática de teste quanto as gramáticas derivadas desta metagramática.

A metagramática de teste (Tabela 1) é uma notação variante da *Extended* BNF (ISO/IEC 14997, 1996) usada para definição da sintaxe de uma linguagem de teste pelo uso de um número de regras. Cada regra nomeia parte da linguagem, denominada símbolo não terminal da linguagem, e define suas possíveis formas (ISO/IEC 14997, 1996). O símbolo terminal de uma linguagem é um átomo que não pode ser dividido em componentes menores da linguagem (ISO/IEC 14997, 1996). Portanto, a metagramática de teste é uma definição formal que nomeia as partes de uma linguagem de teste, mostra que seqüências de símbolos são válidas para compor as sentenças desta linguagem e caracteriza a estrutura de qualquer sentença desta linguagem (ISO/IEC 14997, 1996).

Metagramática de Teste

```
<grammar> = <rule> {<rule>};

<rule> = <non-terminal> '=' <definition> ';'

<definition> = <single-definition> {'|' <single-definition>};

<single-definition> = <term> { ' ' <term> };

<term> = <factor> ['-' <exception>];

<exception> = <factor>;

<factor> = [<multiplicity> '*'] <element>;
```

```
<element> = <repetition>
            | <group>
            | <option>
            | <non-terminal>
            | <terminal>
            | <attribute>
            | <empty>;

<repetition> = '{' <definition> '}';

<group> = '(' <definition> ')';

<option> = '[' <definition> ']';

<non-terminal> = <letter> {<letter> | <decimal-digit>}

<terminal> = "'" <character> - "'" {<character> - "'"} "'"
             | '"' <character> - '"' {<character> - '"'} '"';

<attribute> = '?' {<character> - '?'} '?'

<empty> = ;

<multiplicity> = <decimal-digit> {<decimal-digit>};

/*
Notas:
- todos os símbolos são explicados na seção 3.2.1.;

- os símbolos não terminais <letter>, <decimal-digits> e <character>
não foram definidos;

- a posição de comentários não é definida formalmente;

- comentários são permitidos em qualquer lugar em que vale um caracter
em branco e usam o estilo '/*' ... '*/'.
*/
```

Tabela 1 – Metagramática de Teste

Por ser uma variante de uma notação padrão e por respeitar a sua estrutura, a metagramática de teste herda os benefícios deste padrão (ISO/IEC 14997, 1996). As linguagens derivadas da gramática podem ser definidas objetivamente o que as

tornam de fácil compreensão. A gramática é precisa de modo que as regras não são ambíguas. É formal, permitindo que seja interpretada ou processada por um computador. É natural, sua notação e formato são simples tanto para o aprendizado quanto para a compreensão. É geral, podendo ser aplicada na descrição de diferentes linguagens. Finalmente é auto-descritiva, sendo capaz de descrever a si própria.

As variações em relação à notação *Extended* BNF estão relacionadas à nomenclatura dos elementos da sintaxe e à representação diferente de alguns símbolos terminais. Estas diferenças podem ser evidenciadas ao contrastar sua definição (Tabela 1) com a do padrão (ISO/IEC 14997, 1996).

3.2.1. Definições

Nesta seção são apresentadas as definições utilizadas pela metagramática de teste que foram traduzidas ou adaptadas do padrão que define a notação *Extended* BNF (ISO/IEC 14997, 1996).

Uma ***gramática*** (`grammar`) é constituída de uma ou mais regras gramaticais. As ***regras*** (`rules`) de uma gramática, também denominadas de ***produções***, descrevem o formato das sentenças da linguagem gerada pela gramática, que representam as combinações de elementos do léxico. Toda regra é composta por um identificador (elemento não-terminal) e sua definição, que descreve a estrutura do identificador.

A ***definição*** (`definition`) de um identificador pode ser dada por uma única definição simples ou uma seqüência de alternativas de definições simples (definição múltipla). Cada ***definição simples*** (`single-definition`) equivale a um termo ou uma seqüência de termos. O ***termo*** (`term`) de uma gramática é qualquer seqüência de símbolos representada por um fator, mas não representada por uma exceção. O ***fator*** (`factor`) explicita um número de seqüências de um elemento da sintaxe. Uma ***exceção*** (`exception`) é um conjunto a ser subtraído de um fator e não uma exceção no sentido de controle de fluxo. A ***multiplicidade*** (`multiplicity`) de um fator especifica o número de repetições do elemento associado ao fator.

O elemento (`element`) da sintaxe pode ser uma repetição, um grupo, uma opção, um elemento não-terminal, um elemento terminal, um atributo e um elemento vazio. A ***repetição*** (`repetition`) representa qualquer seqüência de símbolos que pode ser repetida qualquer número de vezes. Quando precedida por uma multiplicidade, este número de vezes é limitado ao que for definido pela multiplicidade. O ***grupo*** (`group`) permite que qualquer seqüência de símbolos seja considerada um elemento da sintaxe. A ***opção*** (`option`) representa uma seqüência opcional de símbolos de uma gramática. Esta seqüência pode ser considerada ou não durante o processamento da regra. Neste segundo caso, apresenta o mesmo valor de uma seqüência vazia. O ***elemento terminal*** (`terminal`) é considerado elemento "primitivo" de uma linguagem e é representado por uma seqüência de caracteres. O ***elemento não-terminal*** (`non-terminal`) é o identificador definido por uma regra gramatical. O ***atributo*** (`attribute`) é um ponto de extensão da gramática e seu significado não é definido por ela. O elemento vazio (`empty`) denota uma seqüência vazia.

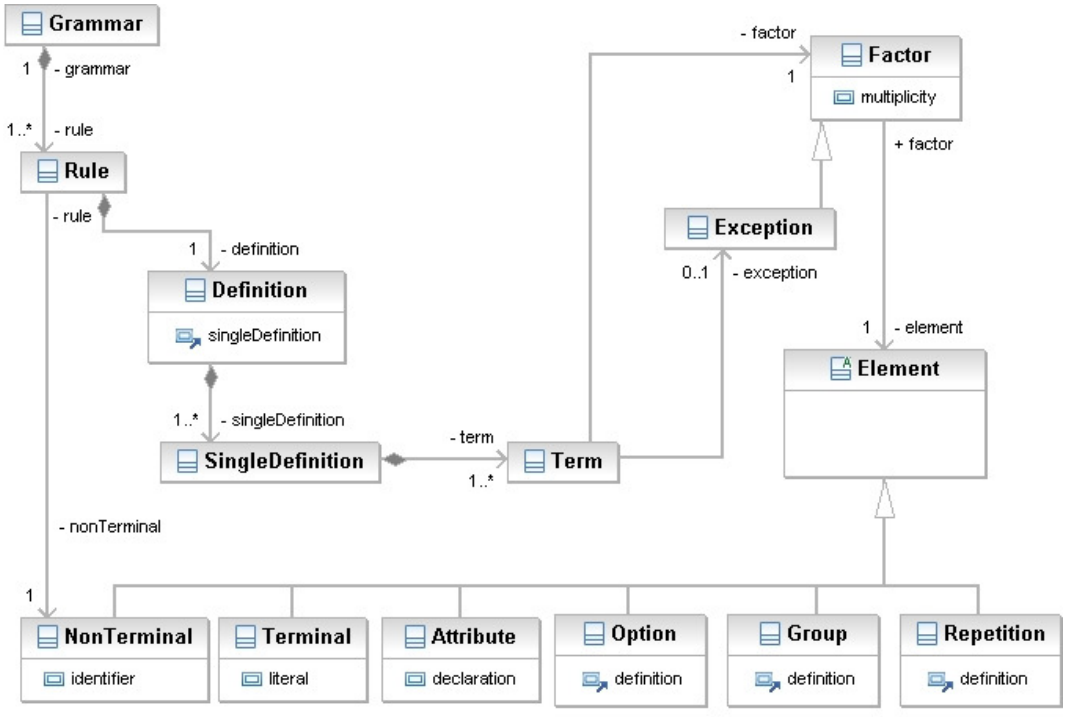

Figura 18 – Diagrama de Classes da Metagramática de Teste

3.3. Semântica das Gramáticas de Teste

A interpretação da estrutura da gramática de teste define a ***semântica*** (significado) de seus elementos. A interpretação é o resultado do processamento direto de elementos terminais ou do processamento de regras gramaticais (produções) correspondentes a cada símbolo identificador a ser interpretado. Como na definição de símbolos pode haver alternação de definições simples, é preciso estabelecer o contexto de aplicação da regra e os critérios de seleção da definição a empregar segundo este contexto. A própria combinação de elementos numa regra pode ser alterada de acordo com este contexto.

Para descrever o contexto e os critérios de seleção, é preciso ampliar esta gramática com uma ***gramática de atributos*** (*Attribute Grammar*). Com o uso de atributos (Duncan & Hutchinson, 1981) é possível definir o fluxo de regras aplicadas para interpretação de um elemento. Eles representarão condições e restrições atribuídas às definições, aos termos e aos elementos da gramática. Tal efeito é obtido com o elemento atributo (`attribute`) descrito nas definições da sintaxe (seção 3.2.1).

Este assunto será abordado novamente quando os modelos de teste forem descritos (seção 3.5). Neste ponto, as semânticas das gramáticas serão tratadas no contexto do teste funcional.

3.4. Trabalhos Relacionados

Alguns trabalhos da literatura de teste de software inspiraram a presente pesquisa e contribuíram para a definição de seu foco e sua abordagem. O uso de gramáticas como modelo de testes e a separação de conceitos independentes e dependentes de plataforma foi uma escolha impulsionada pela análise deste material. Cada subseção aborda a contribuição dos trabalhos para esta pesquisa.

3.4.1. Sistemas Baseados em Gramáticas

O estudo sobre sistemas baseados em gramáticas (Mernik et al., 2004) destaca a importância do uso de gramáticas, particularmente as gramáticas livres

de contexto e as gramáticas de atributos, na implementação de sistemas e na geração automática de ferramentas baseadas em linguagens. Classifica as áreas de aplicação de gramáticas e identifica características de sua aplicação com o intuito de introduzir e popularizar o tema.

Alguns dos principais benefícios apontados sobre o uso de gramáticas são que os sistemas podem se tornar mais genéricos aumentando a sua adaptação a diferentes contextos, o seu desenvolvimento fica mais fácil e sua representação interna pode ser mais eficiente. Esse estudo revela que a maior dificuldade no uso de gramáticas como parte de uma solução está em converter representação do problema para uma forma que faça uso de uma gramática.

Originalmente, as gramáticas foram concebidas como uma notação para a descrição e processamento de uma linguagem. A sua importância para a construção automática de compiladores e interpretadores é reconhecida (Mernik et al., 2004), mas seu emprego não se restringe somente a este tipo de ferramenta. Esse trabalho enumerou ainda alguns tipos de ferramentas geradas a partir de gramáticas. Citou que, na maioria dos casos, as definições principais da linguagem precisam ser estendidas com informações específicas da ferramenta. Noutros casos, somente uma parcela das definições é suficiente para sua geração automática. Destacou ainda que, no processo de construção da ferramenta, há situações onde informações implícitas precisam ser extraídas das definições. Este é o caso de uma sentença da gramática equivalente a uma série de chamadas de métodos durante a execução de um programa.

Ao indicar os geradores de casos de teste como um possível exemplo de ferramenta gerada com base em gramáticas e ao elencar os benefícios e as dificuldades de sistemas baseados em gramáticas, esse trabalhou reforçou a idéia desta pesquisa de utilizar gramáticas na construção de modelos de teste.

3.4.2. Desenvolvimento de Software com Abordagem Gramatical

Um dos aspectos mais importantes no desenvolvimento de software é o correto entendimento dos requisitos do usuário. É necessário compreender o domínio da aplicação a ser construída. Uma abordagem gramatical utilizando um

projeto orientado a objetos foi proposta (Kosar et al., 2003 ou 2004) como solução para entendimento dos requisitos de domínios específicos.

A idéia é identificar e capturar os conceitos e relacionamentos existentes no domínio de um problema por meio de gramáticas livre de contexto e gramáticas de atributos. Para isto foi elaborado um processo. Descreve-se a sintaxe do problema derivando uma gramática livre de contexto de um diagrama conceitual de classes. Depois, descreve-se a semântica do problema associando atributos a todo conceito derivado de casos de uso e de diagramas operacionais usados para retratar cenários. Um protótipo do sistema é construído usando um compilador e a gramática de atributos obtida nos passos anteriores.

Essa proposta influenciou a concepção do modelo estrutural adotado por esta pesquisa. Este modelo focaliza a captura de requisitos que afetam a representação dos dados de teste. Os conceitos e relacionamentos do sistema em teste são capturados e compreendidos segundo o escopo de teste (seção 3.5.1).

3.4.3. Projeto de um Metamodelo Executável Orientado por Gramática

Entendidos os requisitos do sistema, a progressão natural dos esforços de desenvolvimento é a definição de uma arquitetura. Esta arquitetura de software pode ser orientada por uma arquitetura de processos de negócio que está em constante evolução. As mudanças de requisitos ao longo da vida de um sistema são o principal impacto desta evolução.

Com intuito de tornar a arquitetura menos suscetível a essas perturbações, uma linha de pesquisa (Arsanjani, 2005) propõe um projeto de software baseado em gramática aplicado na criação e manutenção de arquiteturas de software reconfiguráveis dinamicamente.

Nesse tipo de projeto, as técnicas de modelagem de processo de negócios são combinadas a arquiteturas orientadas a serviços, linguagens específicas de domínios e metamodelo usado para criar especificações executáveis de processos de negócio. Essas especificações são traduzidas numa gramática do domínio que pode ser empregada tanto no nível de negócio quanto de desenvolvimento. Isto resulta numa representação consistente que facilita a adaptação do sistema ao representar tanto a sua estrutura como o fluxo de processo corrente.

A gramática retratada por esse trabalho serviu de inspiração para a composição das gramáticas que definem o modelo comportamental e o modelo de casos de teste desta pesquisa (seções 3.5.2 e 3.5.3).

3.4.4. Modelos Independentes e Dependentes de Plataforma

Uma solução de um problema ganha escala se é capaz de considerar as particularidades do ambiente onde é empregada. Partindo deste princípio, os modelos propostos para estrutura, para comportamento e para casos de teste devem apresentar características independentes e dependentes da plataforma onde são empregados.

O estudo (Heckel & Lohmann, 2003) sobre testes dirigidos por modelos se fundamentou nesta orientação para conseguir executar testes em plataformas diferentes sem desperdiçar o tempo dedicado à construção de uma lógica comum de teste de sistemas que não é afetada pelo ambiente.

Esse trabalho conseguiu desacoplar as partes dependentes e independentes de plataforma por meio de padrões de projeto de software aplicados ao modelo de teste em questão. Embora o modelo de teste usado não seja gramatical, o valor deste resultado para esta pesquisa está relacionado ao fato de que a separação de conceitos deve ser concebida na etapa de elaboração dos modelos de teste.

3.5. Modelos de Teste

A partir da definição de uma metagramática de teste e do conhecimento adquirido da análise dos trabalhos relacionados, esta pesquisa propõe os modelos gramaticais, descritos nas próximas subseções. Estes modelos representam interpretações de gramáticas de teste, que foram concebidas pra serem representações mínimas o suficiente para caracterizar o sistema em teste, a estrutura de teste e o ambiente de teste.

A semântica de cada gramática de teste está de acordo com o propósito da modelagem. A visão da estrutura e do comportamento do sistema de teste, dos casos de teste e do ambiente de teste do sistema são propósitos de modelagem identificados durante a sistematização do teste funcional. Todos os modelos

foram concebidos para atender estes propósitos e foram elaborados com base nas especificações de requisitos funcionais do sistema em teste e do ambiente de teste. Retratam os aspectos observados via interface pública do sistema em teste e via parâmetros de configuração do ambiente em teste.

Para facilitar a compreensão da aplicação dos modelos gramaticais, um pequeno exemplo foi projetado com o objetivo de capturar os principais conceitos de um projeto orientado a objetos. Trata-se de um sistema simples de manipulação de contas bancárias inspirado no exemplo apresentado pelo *Unified Modeling Language Testing Profile* (UML2TP, 2003).

Nesse sistema (Figura 19), existem os conceitos: pessoa física (*Person*) e organização (*Organization*), que são espécies de parte (*Party*); pessoa jurídica (*Legal Organization*), que é uma espécie de organização; banco (*Bank*), que é uma espécie de pessoa jurídica. O banco é composto por agências. Cada agência (*Branch*) mantém contas correntes associadas. Cada conta corrente (*Account*) está associada a uma parte. O seu saldo é manipulado por operações de crédito e débito. O serviço bancário (*Bank Service*) permite a transferência de fundos (quantias) entre contas e mantém uma relação dos bancos existentes e das transferências realizadas. Cada transferência (*Account Transfer*) registra a conta de origem e a conta de destino, a data de ocorrência e a quantia movimentada.

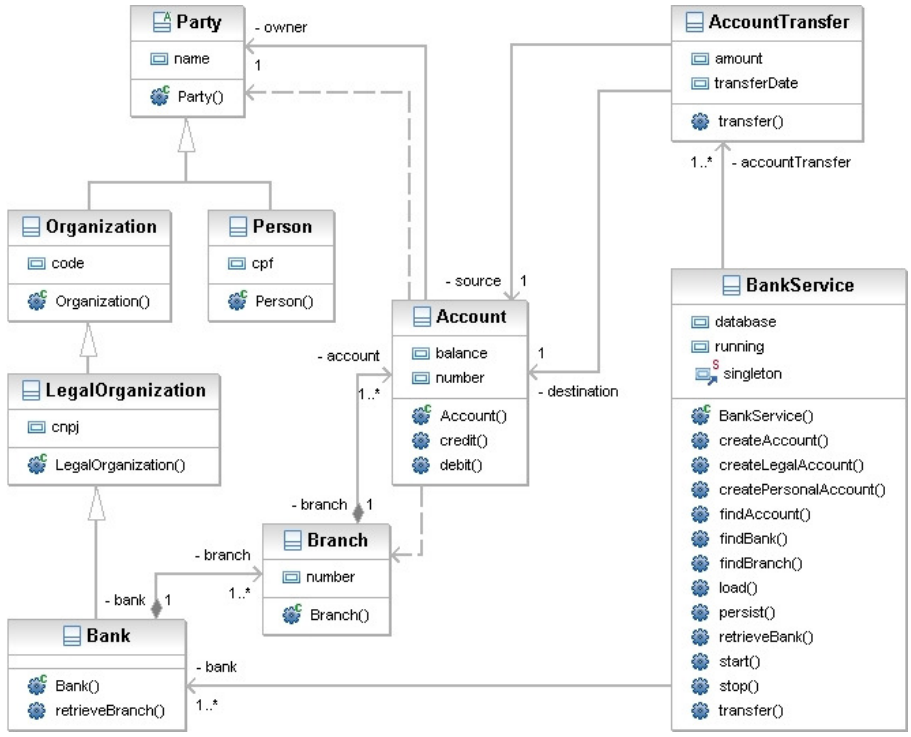

Figura 19 – Exemplo de Sistema em Teste

3.5.1.
Modelo Estrutural do Sistema em Teste

O objetivo do modelo estrutural do sistema em teste, ou modelo estrutural (*structural model*), é descrever os conceitos deste sistema e os relacionamentos entre estes conceitos. Ele é descrito segundo a gramática de estrutura (Tabela 2).

Gramática de Estrutura

```
<sut> = <type> {<type>};                    /* sut = system under test */

<type> = <program-unit> | <primitive-type> | <enumeration>;

<program-unit> = <program-unit-id> '=' <program-unit-definition> ';';
<program-unit-id> = <non-terminal>;
<program-unit-definition> = [ ( {<super-unit>} ) ] ( {<property>} );

<super-unit> = <non-terminal>;

<property> = <property-type> [ <property-id> ];
<property-type> = <factor>;
<property-id> = ':' <non-terminal>;

<primitive-type> = '$' <primitive-id> <primitive-definition>;
<primitive-id> = <non-terminal>;
<primitive-definition> = <empty>;

<enumeration> = <enumeration-id> '=' <enumerarion-definition> ';';
<enumeration-id> = <non-terminal>;
<enumerarion-definition> = <literal> {<literal>};
<literal> = <terminal>;

/* Nota: os símbolos <empty>, <terminal>, <non-terminal> e <factor> são
definidos na metagramática de teste. */
```

Tabela 2 – Gramática de Estrutura

Esta gramática define a parte estrutural do modelo representado pelo sistema em teste (`sut`). Ela define cada tipo (`type`) declarado neste modelo. Os tipos possíveis são a unidade de programação, o tipo primitivo e a enumeração.

Nesta gramática, uma unidade de programação (`program-unit`) é composta por propriedades e mantém referências a outras unidades de programação. Cada propriedade (`property`) apresenta um identificador e uma referência a um tipo. Cada referência a outra unidade de programação (`super-unit`) revela uma relação de dependência na sua definição.

Um tipo primitivo (`primitive-type`) é composto de um identificador e, por ser um elemento básico, não apresenta definição. Uma enumeração (`enumeration`) possui um identificar e é composta por uma coleção de literais. Cada literal (`literal`) é um elemento terminal da linguagem.

A representação da interpretação desta gramática gera o seguinte modelo computacional (Figura 20).

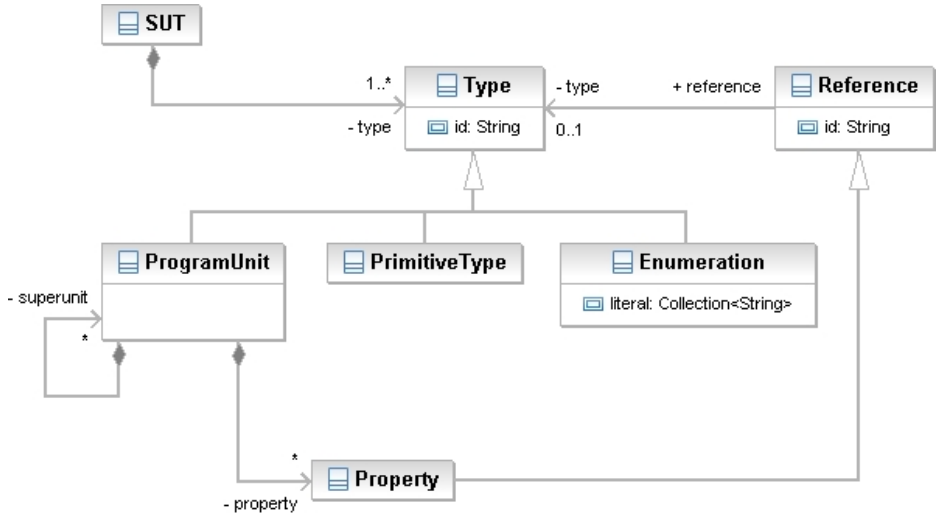

Figura 20 – Representação do Modelo Estrutural

No paradigma de orientação a objeto, cada ***classe*** é representada por uma unidade de programação, cada ***tipo primitivo*** e cada ***enumeração*** são representados pelos tipos correspondentes no modelo. O ***atributo simples*** de uma classe corresponde à propriedade com tipo definido por um fator composto de um elemento não-terminal.

Segundo este mesmo paradigma, o relacionamento de ***herança*** ou ***generalização*** é definido por cada referência mantida pela unidade de programação a outra unidade de programação. O relacionamento de ***associação***, ***composição*** ou ***agregação*** entre unidades de programação é definido por cada propriedade da unidade de programação. Se o tipo da propriedade for definido por um fator composto de um elemento não-terminal, então equivale a uma

associação simples de multiplicidade um `(1)`; se for definido por um fator igual a uma opção composta de um elemento não terminal, equivale a uma ***associação simples de multiplicidade zero ou um*** `(0..1)`; ser for definido por uma repetição de elemento não terminal, equivale a uma ***composição ou agregação de multiplicidade ilimitada*** `(*)`; se for definido por um fator com multiplicidade `1*` e composto por uma repetição de um elemento não terminal, equivale a uma ***composição ou agregação de multiplicidade um ou ilimitada*** `(1..*)`. Um mapa de elementos é obtido se cada repetição usada nas associações, composições ou agregações citadas for constituída por um elemento não terminal (chave) e outro elemento (valor).

No exemplo do sistema de contas bancárias, o modelo estrutural descreve todas as classes do sistema descritas anteriormente (seção 3.5), as hierarquias de classes existentes (herança), suas propriedades e seus relacionamentos (Tabela 3).

Modelo Estrutural do Exemplo

```
BankService = () ({Bank}:bank
                  {AccountTransfer}:accountTransfer);

Bank = (LegalOrganization) ({Branch}:branch);

LegalOrganization = (Organization) ($string:code);

Organization = (Party) ($string:code);

Party = ($string:name);

AccountTransfer = (Account:source Account:destination
                   $double:amount $date:transferDate);

Account = ($string:number $double:balance
           Party:owner Branch:branch);

Branch = ($string:number Bank:bank {Account}:account);

Person = (Party) ($string:cpf);
```

Tabela 3 – Modelo Estrutural do Exemplo

3.5.2.
Modelo Comportamental do Sistema em Teste

O objetivo do modelo comportamental do sistema em teste, ou modelo comportamental (*behavior model*), é descrever as funcionalidades e os serviços oferecidos por este sistema via sua interface pública. Ele é descrito segundo a gramática de comportamento (Tabela 4).

Gramática de Comportamento

```
<sut> = <type> {<type>};

<type> = <program-unit>;

<program-unit> = <program-unit-id> '=' <program-definition> ';';
<program-unit-id> = <non-terminal>;
<program-unit-definition> = ( { <operation> } );

<operation> = <operation-id> ( [ { <operation-parameter> } ] )
              <operation-type> [ { <operation-exception> } ] ';';
<operation-id> = <non-terminal>;
<parameter> = <parameter-type> [ <parameter-id> ];
<parameter-type> = <factor>;
<parameter-id> = ':' <non-terminal>;
<operation-type> = <factor>;
<operation-exception> = <non-terminal>;

/* Nota: os símbolos <non-terminal> e <factor> são definidos na
metagramática. */
```

Tabela 4 – Gramática de Comportamento

Esta gramática define a parte estrutural do modelo representado pelo sistema em teste (`sut`). Ela define cada tipo (`type`) declarado neste modelo. Neste caso o único tipo possível é a unidade de programação.

Nesta gramática, uma unidade de programação (`program-unit`) é composta por operações. Cada operação (`operation`) possui um identificador, um grupo de parâmetros, um tipo e exceções possivelmente geradas. Cada parâmetro (`parameter`) possui um identificador e um tipo. O tipo do parâmetro

e da operação é interpretado da mesma forma que o tipo de propriedade do modelo estrutural.

A representação da interpretação desta gramática complementa o modelo computacional (Figura 21).

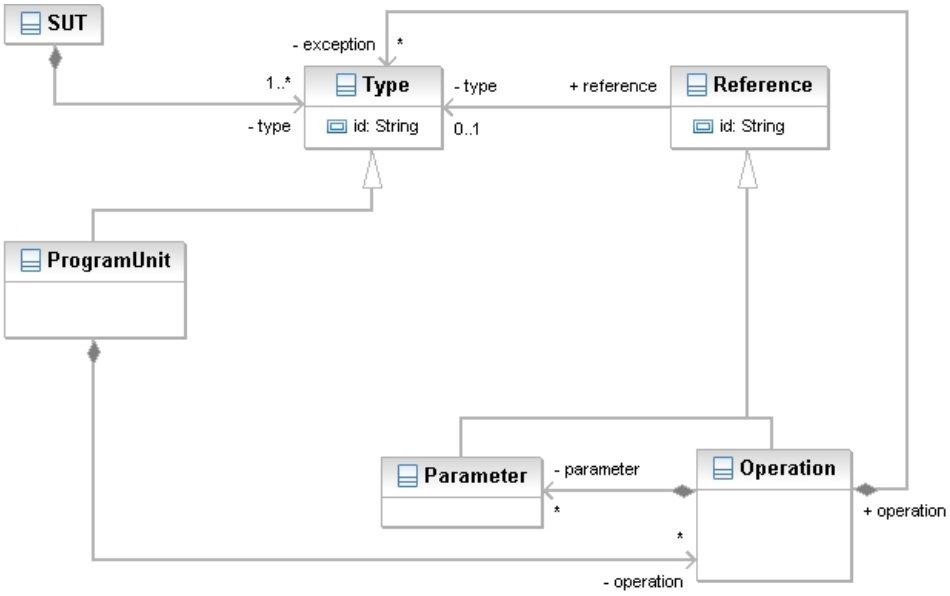

Figura 21 – Representação do Modelo Comportamental

O modelo completo do sistema em teste é definido pela composição do modelo estrutural e do modelo comportamental (Figura 22).

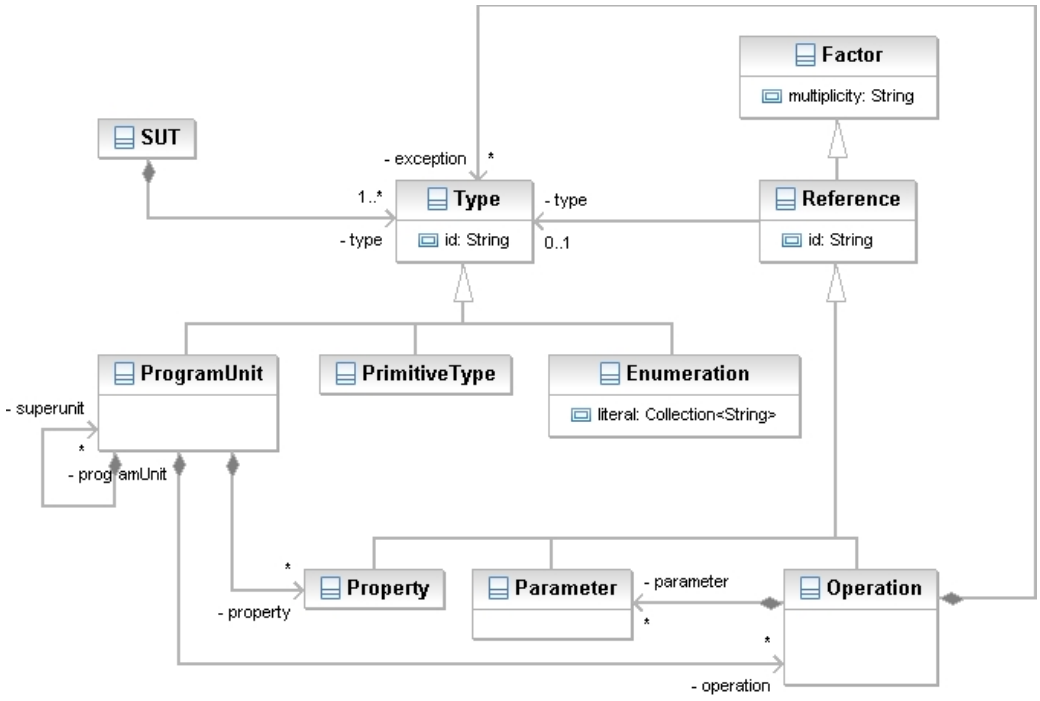

Figura 22 – Modelo Completo do Sistema em Teste

No paradigma de orientação a objetos, as operações correspondem à declaração dos métodos das classes representadas pelas unidades de programação.

No exemplo do sistema de contas bancárias, o modelo comportamental descreve as operações de crédito e débito de quantias, recuperação e definição de saldo de uma conta corrente (Tabela 5).

Modelo Comportamental do Exemplo

```
Account = ( makeAccount ($string:number Branch:branch
                         Party:owner $double:balance) Account;
            credit ($double:amount) $void IllegalAmountException;
            debit ($double:amount) $void
                   IllegalAmountException InsufficientFundsException;
            getBalance () $double;
            getBranch () Branch;
            getNumber () $string;
            getOwner () $string;
            setBalance( $double:balance ) $void IllegalAmountException;
            setBranch( Branch:branch ) $void;
            setNumber( $string:number ) $void;
            setOwner( Party:owner ) $void;
            toString $string; );

Branch = ( makeBranch ($string:number, Bank:bank) Branch;
           getAccount {Account}; );

Bank = ( makeBank ($string:name $string:code $string:cnpj) Bank; );

Person = ( makePerson( $string:name ) Person; );

/* Notas: somente as operações que serão avaliadas foram declaradas;
as exceções das operações não foram declaradas e só estão representadas
para completar o exemplo de descrição. */
```

Tabela 5 – Modelo Comportamental do Exemplo

3.5.3. Modelo de Casos de Teste

O objetivo do modelo de casos de teste (*test case model*) é descrever o conjunto de casos de testes a serem executados com a intenção de avaliar as funcionalidades de um sistema. Cada caso de teste estabelece um fluxo de

processamento para exercitar o sistema em teste e o seu resultado esperado. Este modelo é descrito segundo a gramática de casos de teste (Tabela 6).

Gramática de Casos de Teste

```
<test-suite> = <test-case> {<test-case>};

<test-case> = <test-goal> <test-oracle>;

<test-goal> = <test-goal-id> '=' <test-goal-definition> ';';
<test-goal-id> = 'testGoal';
<test-goal-definition> = [ <block> ] <test-result>;

<test-oracle> = <test-oracle-id> '=' <test-oracle-definition> ';';
<test-oracle-id> = 'testOracle';
<test-oracle-definition> = [ <block> ] <test-result>;

<block>     = ( {<statement>} );
<statement> = [<variable> '='] <instance> ';';
<instance>  = <constant> | <variable> | <file> | <call>;
<constant>  = [<test-constraint>] <terminal>;
<variable>  = [<test-constraint>] <attribute>;
<file>      = [<test-constraint>] '@' <filename>;
<filename>  = <nonterminal>;
<call>      = [<test-constraint>]
              [<context>] <call-id> ( {<argument>} ) ';';
<context>   = <variable>;
<call-id>   = <non-terminal>;
<argument>  = <factor>;

<test-constraint> = <attribute>;
<test-result> = <instance>;

/* Nota: os símbolos <element>, <non-terminal>, <terminal> e
<attribute>, <factor> são definidos na metagramática. */
```

Tabela 6 – Gramática de Casos de Teste

Esta gramática define o modelo representado pelo conjunto de teste do sistema (`test-suite`). Ela define cada caso de teste (`test-case`) declarado neste modelo como uma composição de objetivo de teste (`test-goal`) e oráculo de teste (`test-oracle`).

Nesta gramática, cada objetivo de teste e cada oráculo de teste correspondente apresentam identificadores constantes, respectivamente, `testGoal` e `testOracle`. Ambos são compostos por um bloco de comandos e por um resultado.

O bloco de comandos (`block`) é um grupo de comandos disparados ou não contra o sistema em teste. Cada comando (`statement`) é composto opcionalmente pela declaração de uma variável e obrigatoriamente pela definição de uma instância. A variável (`variable`) é um atributo da gramática de teste onde é armazenado o resultado de um processamento. A instância (`instance`) equivale ao resultado de um processamento, podendo ser uma constante, uma variável, um arquivo ou uma chamada. A constante (`constant`) é um símbolo terminal. A chamada (`call`) dispara uma operação descrita no modelo do sistema em teste. Ela é composta de um identificador equivalente ao identificador da operação disparada e por um grupo de argumentos. Cada argumento (`argument`) tem correspondência com o parâmetro definido pela operação disparada. A chamada é feita num contexto (`context`), que é representado por uma variável. Quando a instância é um arquivo (`file`), então o resultado de um processamento é armazenado no formato de um arquivo de texto nomeado (`filename`).

O resultado do objetivo de teste equivale ao resultado obtido do caso de teste. O resultado do oráculo de teste equivale ao resultado esperado para este mesmo caso de teste. O resultado (`result`) é uma instância.

Casos de teste mais complexos podem apresentar um controle de fluxo por meio de condições e restrições (`test-constraint`) atribuídas aos elementos da gramática. Estas restrições são representadas por atributos associados a cada tipo de instância da gramática de teste. Estes atributos estabelecem critérios de seleção (ramificações) e execução (iterações) de regras de acordo com informações de contexto. São utilizados com semânticas diferentes de acordo com a estratégia de teste aplicada no seu processamento. Uma descrição que utiliza este artifício torna a compreensão do caso de teste mais complexa na medida em que um caso de teste acaba correspondendo a vários “subcasos” de teste.

A representação da interpretação desta gramática gera o seguinte modelo computacional (Figura 23).

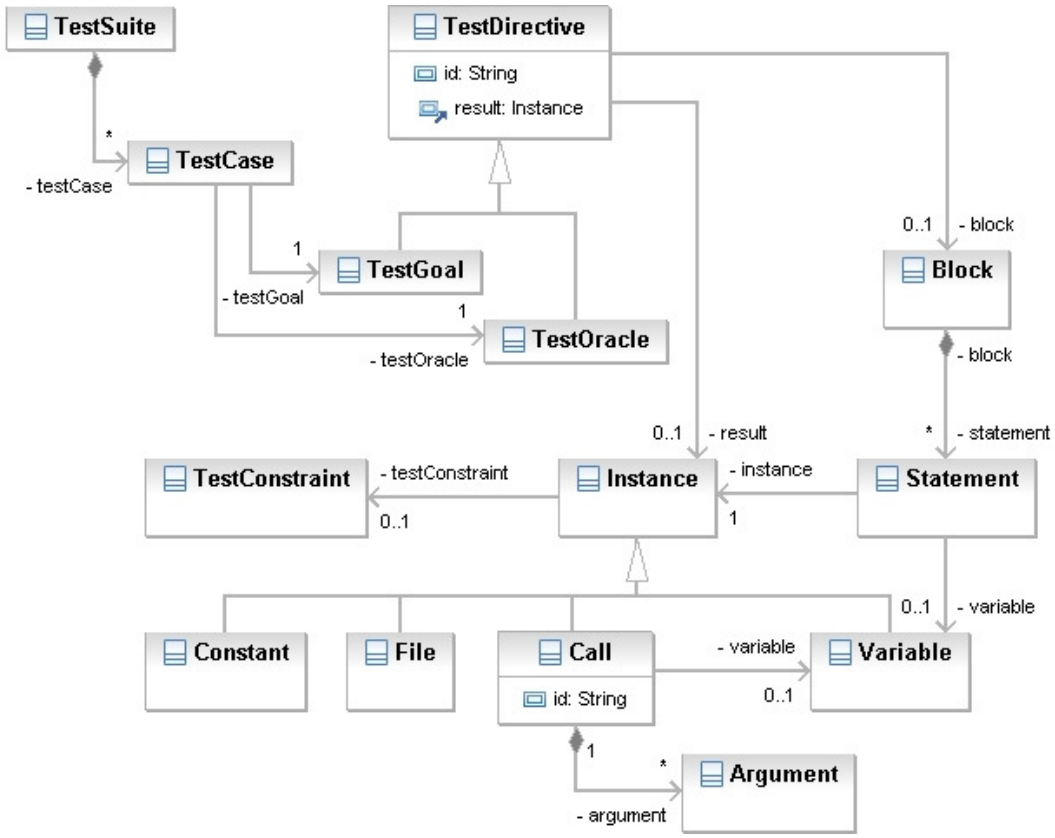

Figura 23 – Representação do Modelo de Casos de Teste

No paradigma de orientação a objetos, o bloco de comandos corresponde ao bloco de comandos de um programa. Cada objetivo de teste e cada oráculo de teste correspondem ao corpo de métodos. Cada caso de teste corresponde a um método que compara os resultados de um objetivo de teste e de um oráculo de teste. O conjunto de teste equivale a um modelo com uma unidade de programação que abriga todos estes métodos.

No exemplo do sistema de contas bancárias, o modelo de casos de teste descreve casos de teste em que o saldo de uma conta corrente é manipulado por operações de crédito e débito, sendo o resultado esperado ora definido diretamente no valor de uma quantia ora calculado por meio de outros métodos (Tabela 7).

Modelo de Casos de Teste do Exemplo

`testGoal = ( makeAccount( 1234 aBranch aPerson 0 ) ?createdAccount?;`
`?createdAccount? credit( 10 );`
`) ?createdAccount? getBalance();`
`testOracle = 10;`

```
/* Notas:
- este exemplo avalia a operação de crédito;
- este caso de teste considera que o construtor de conta e o método de
acesso ao saldo (balance) foram avaliados com sucesso em casos de teste
anteriores.  Do contrário, o teste proposto poderia ser mascarado por
um defeito nestas operações;
- o exemplo é simples e mostra como pode ser realizado o processamento
do objetivo de teste e como um resultado esperado pode ser definido;
- atributos (ex. ?createdAccount?) são utilizados para armazenar os
valores intermediários de processamento e para definir o contexto da
chamada de uma operação. */
```

Tabela 7 – Modelo de Casos de Teste do Exemplo

3.5.4. Modelo do Ambiente de Teste

O objetivo do modelo do ambiente de teste (*test environment model*) é estabelecer o estado inicial do sistema em teste e definir as configurações do ambiente específicas de plataforma que afetam a geração e a execução dos casos de teste deste sistema. Este modelo é descrito pela gramática de ambiente de teste (Tabela 8).

Gramática de Ambiente de Teste

```
<test-environment> = <env-statement> {<env-statement>};

<env-statement> = <mapping> | <startup>;

<mapping> = <mapping-key> '=' <mapping-value> ';';
<mapping-key> = <non-terminal>;
<mapping-value> = <terminal> {<terminal>};

<startup> = 'startup' <block>;

/* Nota: os símbolos <terminal> e <non-terminal> são definidos na
metagramática.  O símbolo <block> é definido na gramática de casos de
teste. */
```

Tabela 8 – Gramática de Ambiente de Teste

Esta gramática define o modelo representado pelo ambiente de teste (`test-environment`). Ela define cada comando de ambiente que pode ser ou um mapeamento ou uma seqüência de comandos de inicialização.

Nesta gramática, um mapeamento (`mapping`) é constituído por par chave/valor que relaciona os elementos dos modelos de teste com conceitos dependentes da plataforma de realização dos testes. A chave (`mapping-key`) corresponde a um identificador utilizado no modelo e o valor (`mapping-value`) corresponde à descrição, por meio de terminais, de um parâmetro de configuração da plataforma de teste.

Além do mapeamento, existe a seqüência de comandos de inicialização (`startup`). Esta seqüência equivale a um bloco de comandos, como definido no modelo de casos de teste, responsáveis por conduzir o sistema de teste a um estado inicial bem definido.

A representação da interpretação desta gramática gera o seguinte modelo computacional (Figura 24).

Figura 24 – Representação do Modelo de Ambiente de Teste

No paradigma de orientação a objetos, esta gramática define como será feita a instanciação de classes do domínio definido pelo modelo estrutural (seção 3.5.1) ou a execução eventualmente necessária de métodos definidos pelo modelo comportamental (seção 3.5.2). A interpretação de suas regras permitirá atingir um estado de referência do sistema que servirá como base de realização dos testes.

O modelo de ambiente procura garantir o contexto inicial utilizado pelos casos de teste de modo a permitir que os mesmos possam ser novamente executados em condições idênticas ou equivalentes. Isto favorece a análise dos casos de teste em relação a possíveis mudanças de requisitos e está alinhado com a idéia de testes de regressão.

As configurações do ambiente são igualmente retratadas por regras gramaticais que definem valores para variáveis de ambiente que deverão ser configuradas pelo dispositivo de teste dependente de uma plataforma específica.

No exemplo do sistema de contas bancárias, o modelo de ambiente de teste (Tabela 9) descreve os objetos que devem ser criados no início da execução do sistema em teste. Para citar alguns destes objetos: pessoa física, banco, agência e conta corrente. Também define como estão mapeados os conceitos na plataforma de execução, que neste exemplo é retratada usando a linguagem Java (*Sun Microsystems*).

Modelo de Ambiente de Teste do Exemplo

```
startup = ( aPerson = makePerson ('Silva' '000.000.000-00');
            aBank = makeBank ('Banco' '001' '00.000.000/0000-00' );
            aBranch = makeBranch ('1000' aBank) );

/* Mapeamento da estrutura para a plataforma Java */

Person = sample.Banking.Person;
Bank = sample.Banking.Bank;
Branch = sample.Banking.Branch;
/* ... (continua) ... */

/* Mapeamento dos comportamentos para a plataforma Java */

Person.makePerson = Person;
Account.makeAccount = Account;
Account.credit = credit;
Account.debit = debit;
Account.getBalance = getBalance;
Account.setBalance = setBalance;
/* ... (continua) ... */

/* Nota: cada operação é definida por uma chave composta de nome da
unidade de programação, '.' e nome da operação.  O valor da chave
equivale ao nome do método correspondente na plataforma Java. */
```

Tabela 9 – Modelo de Ambiente de Teste do Exemplo

Todos os modelos apresentados compõem o cenário de teste funcional representado no seguinte modelo computacional (Figura 25).

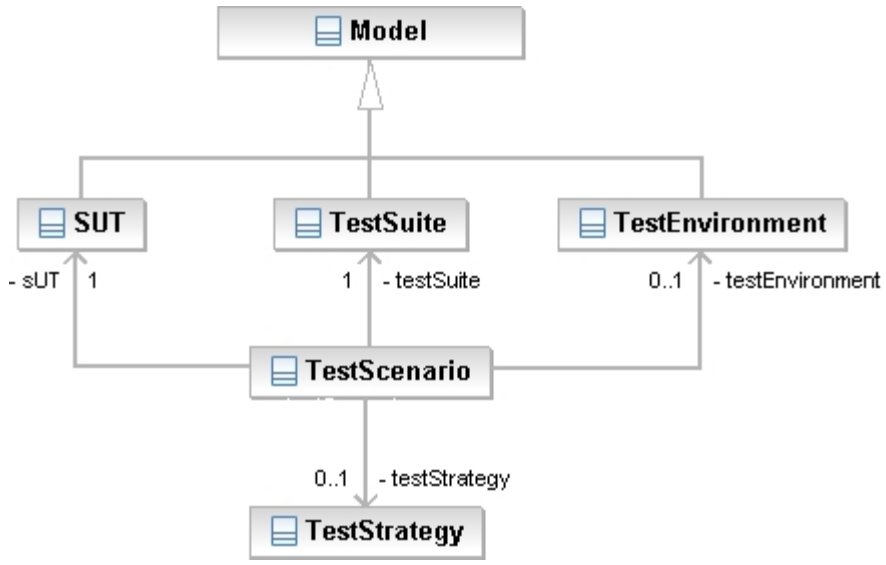

Figura 25 – Representação do Cenário de Teste