INPE-00000-TDI/0000

J-SDL: UM FRAMEWORK PARA A SIMULAÇÃO DE ESPECIFICAÇÕES EM SDL E GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE

Júlio Resende Ribeiro

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Prof. Dr. Nandamudi L. Vijaykumar

INPE
São José dos Campos
2008

Publicado por:

Instituto Nacional de Pesquisas Espaciais (INPE)
Gabinete do Diretor – (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 – CEP 12.245-970
São José dos Campos – SP – Brasil
Tel.: (012) 3945-6911
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

Solicita-se intercâmbio
We ask for exchange

INPE-00000-TDI/0000

J-SDL: UM FRAMEWORK PARA A SIMULAÇÃO DE ESPECIFICAÇÕES EM SDL E GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE

Júlio Resende Ribeiro

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Prof. Dr. Nandamudi L. Vijaykumar

INPE
São José dos Campos
2008

Dados Internacionais de Catalogação na Publicação

Cutter

Ribeiro, Júlio Resende.
J-SDL: Um framework para simulação de especificações em SDL e geração automática de casos de teste
São José dos Campos: INPE, 2008.
00p. ; (INPE-0000 -TDI/00)

1. SDL. 2. Simulação. 3. Especificação.
4. Verificação e Validação. 5.Casos de Teste. I. J-SDL: Um framework para simulação de especificações em SDL e geração automática de casos de teste

"Há um tempo em que é preciso abandonar as roupas usadas, que já tem a forma do nosso corpo, e esquecer os nossos caminhos, que nos levam sempre aos mesmos lugares. É o tempo da travessia: e, se não ousarmos fazê-la, teremos ficado, para sempre, à margem de nós mesmos."

Fernando Pessoa

Aos meus pais: Antônio Tadeu Vieira Ribeiro e Lylia Resende Ribeiro.

AGRADECIMENTOS

Agradeço em primeiro lugar aos meus pais: Antônio Tadeu Vieira Ribeiro e Lylia Resende Ribeiro, por tudo que fizeram por mim, a vida toda, priorizando sempre e em todos os aspectos a realização dos meus objetivos. É deles, grande parte dessa conquista.

Agradeço muito ao meu orientador: Dr. Prof. Nandamudi Lankalapali Vijaykumar pela disposição em me orientar, com muita paciência e dedicação durante a realização desse trabalho. Da mesma forma, agradeço ao Prof. Valdivino Alexandre de Santiago Júnior que teve uma importante participação para o direcionamento e andamento desse trabalho.

Gostaria de agradecer a todos os meus amigos da FAI – Faculdade de Administração e Informática de Santa Rita do Sapucaí MG, em especial a alguns professores que durante a graduação desempenharam um papel primordial na minha formação profissional e sempre me incentivaram a seguir caminhos mais desafiadores: Prof. Aldo Ambrósio Morelli, Profa. Eunice Gomes de Siqueira, Prof. Fábio Gavião de Avelino Neto, Prof. Dr. Nilson Sant'Anna, Prof. Roberto Souza Porto e Profa. Sra. Silvana Isabel de Lima.

Agradeço aos amigos Alexandre Franco de Magalhães e Valeska Pivoto Patta Marcondes pelo apoio e pelo companheirismo ao longo de várias viagens a São José dos Campos.

Sou grato também aos meus alunos, com os quais aprendo muito a cada dia, em especial à aluna de iniciação científica Fabiana Fraga Ferreira.

Por último, mas de forma não menos importante, agradeço a todos os colegas e amigos do programa de pós-graduação em computação aplicada do INPE. Nunca vou me esquecer deles.

RESUMO

A finalidade dos testes de software é detectar faltas latentes em um produto de software antes que o mesmo seja colocado em funcionamento. Há métodos que geram casos de testes desde que o software esteja representado por uma Máquina de Estados (MEF). Devido a algumas limitações das MEFs, investiga-se técnicas formais que permitam uma representação explícita de atividades paralelas e encapsulamento muito necessário nos softwares modernos. Já foram exploradas para este objetivo técnicas a partir de Statecharts que consiste em converter a sua representação para uma MEF a partir da qual se possa extrair casos de testes. Este trabalho apresenta uma abordagem para geração automática de casos de teste, de forma direta (dispensando a transformação para maquina de estados finitos). Essa abordagem consiste na utilização de um framework, desenvolvido para permitir a simulação de comportamento, em termos de mudança de estados, a partir de especificações em SDL, denominado J-SDL.

J-SDL: A FRAMEWORK FOR SIMULATION OF SPECIFICATIONS IN SDL AND AUTOMATIC GENERATION OF TEST CASES

ABSTRACT

The objective of tests in software is to detect faults before it can be released. Methods exist to generate test cases as long as the software behavior is represented as a Finite State Machine (FSM). Due to some limitations in FSM, it is necessary to investigate specification techniques that enable explicit representation of parallel activities and hierarchy very much necessary to model modern software. This has already been explored by using Statecharts in which the Statecharts representation is converted into a FSM from which test cases are generated. The approach used in this dissertation is to investigate the feasibility of using SDL to generate automatically test cases in a straightforward manner, i.e., without the necessity to convert into a FSM. A framework J-SDL has been developed to simulate the behavior in terms of state changes and by simulating this behavior it will be shown that test cases can be generated.

SUMÁRIO

Pág.

LISTA DE FIGURAS29
LISTA DE TABELAS
LISTA DE SIGLAS E ABREVIATURAS
1 INTRODUÇÃO 27
2 SDL 33
2.1 Conceitos básicos 34
2.2 Elementos Principais 37
2.2.1 Elementos de estrutura 37
2.2.2 Elementos de comunicação 42
2.2.3 Elementos de comportamento 44
3 O FRAMEWORK J-SDL 49
3.1 Visão de Análise 50
3.1.1 RF01 - Suporte a elementos básicos de estrutura 51
3.1.2 RF02 - Suporte a elementos básicos de comunicação 54
3.1.3 RF03 - Suporte a elementos básicos de comportamento 55
3.1.4 RF04 - Consultar o estado atual da simulação 57
3.1.5 RF05 - Consultar número de ativações por elemento Input 58
3.1.6 RF06 - Consultar número de ativações por elemento State 58
3.1.7 RF07 - Estimular sinais 58
3.1.8 RF08 - Reiniciar a simulação 58
3.1.9 RNF01 - Independência de plataforma 59
3.1.10 RNF02 - Independência de software comercial. 59
3.1.11 RNF013 – Possibilitar expansão e adaptação 59
3.2 Visão de Projeto 59
3.2.1 O Pacote *Model* 61
3.2.2 O Pacote *Service* 65
4 RESULTADOS 67
4.1 Estudo de Caso: Sistema de Manufatura 69

4.2 Estudo de Caso: Sistema Produtor/Consumidor 75
5 CONCLUSÃO E TRABALHOS FUTUROS 81
REFERÊNCIAS BIBLIOGRÁFICAS 85
BIBLIOGRAFIA RECOMENDADA 87

LISTA DE FIGURAS

Pág.

Figura 1.1 – Relacionamento entre os conceitos: falta, erro e falha. 29
Figura 2.1 – Interação entre ambiente e sistema.....34
Figura 2.2 – Particionamento em níveis de abstração.....35
Figura 2.3 – Utilização de tipos, instâncias e conjuntos. 36
Figura 2.4 – Definição de níveis de sistema.....38
Figura 2.5 – Declaração de tipos de bloco. 38
Figura 2.6 – Definição de níveis de bloco.....39
Figura 2.7 – Instância de blocos e conjuntos de blocos.....39
Figura 2.8 – Definição de sistema através de instâncias de bloco.....40
Figura 2.9 – Declaração de tipos de processo. 40
Figura 2.10 – Declaração de tipos de bloco. 41
Figura 2.11 – Instância de processos e conjuntos de processos.....41
Figura 2.12 – Definição de tipo de bloco através de instâncias de processo. 42
Figura 2.13 – Definição de canais. 43
Figura 2.14 – Exemplo de utilização de canais. 44
Figura 2.15 – Representação do elemento *State.* 45
Figura 2.16 – Representação do elemento *Start.* 45
Figura 2.17 – Representação do elemento *Input.*.....46
Figura 2.18 – Representação do elemento *Output.* 46
Figura 2.19 – Representação do elemento *Save*. 46
Figura 2.20 – Representação do elemento *Decision*. 47
Figura 2.21 – Exemplo de definição de comportamento. 48
Figura 3.1 – Comportamento dos elementos *System* e *Block.* 52
Figura 3.2 – Comportamento do elemento *Process.* 53
Figura 3.3 – Ambiente típico de utilização do J-SDL. 60
Figura 3.4 – Diagrama de pacotes referente aos módulos do J-SDL. 61
Figura 3.5 – Diagrama de pacotes referente ao pacote *Model*. 62
Figura 3.6 – Diagrama de classes referente ao pacote S*tructure* 62
Figura 3.7 – Diagrama de classes referente ao pacote *Communication* 64
Figura 3.8 – Diagrama de classes referente ao pacote *Behavior* 65
Figura 3.9 – Diagrama de classes referente ao pacote *Service.* 66
Figura 4.1 – Critério de cobertura por elemento Input. 67
Figura 4.2 – Critério de cobertura por elemento Input. 68
Figura 4.3 – Definição do nível de sistema do Sistema de Manufatura. 69
Figura 4.4 – Definição do bloco Principal do Sistema de Manufatura. 71
Figura 4.5 – Definição do processo Máquina1 do Sistema de Manufatura. 72
Figura 4.6 – Definição do processo Máquina2 do Sistema de Manufatura. 72
Figura 4.7 – Definição do processo Supervisor do Sistema de Manufatura. 74
Figura 4.7 – Definição do bloco Principal do Sistema Produtor/Consumidor. 76
Figura 4.8 – Definição do bloco Principal do Sistema Produtor/Consumidor. 77
Figura 4.9 – Definição do processo Produtor do Sistema Produtor/Consumidor. 78
Figura 4.10 – Definição do processo Produtor do Sistema Produtor/Consumidor. 79
Figura 4.11 – Definição do processo *Buffer* do Sistema Produtor/Consumidor. 79

LISTA DE TABELAS

Pág.

Tabela 3.1 – Requisitos funcionais do J-SDL. 50
Tabela 3.2 – Requisitos não funcionais do J-SDL. 51
Tabela 3.3 – Atributos dos elementos *System* e *Block*. 51
Tabela 3.4 – Atributos do elemento *Process*. 52
Tabela 3.5 – Atributos do elemento *Gate*. 54
Tabela 3.6 – Atributos do elemento *Channel*. 54
Tabela 3.7 – Atributos do elemento *Signal*. 55
Tabela 3.8 – Atributos do elemento *State*. 56
Tabela 3.9 – Atributos do elemento *Input*. 56
Tabela 3.10 – Atributos do elemento *Output*. 57
Tabela 3.11 – Atributos do elemento *Save*. 57
Tabela 4.1 – Sinais utilizados na especificação do Sistema de Manufatura. 70
Tabela 4.2 – Estados do Sistema de Manufatura. 73
Tabela 4.3 – Casos de teste para o Sistema de Manufatura. 75
Tabela 4.4 – Sinais utilizados na especificação do sistema Produtor/Consumidor. 76
Tabela 4.5 – Estados do Sistema de Manufatura. 78
Tabela 4.6 – Casos de teste para o Sistema Produtor/Consumidor. 80

LISTA DE SIGLAS E ABREVIATURAS

CCITT	Comitê Consultatif International Télégraphique et Téléphonique
CEA	Ciências Espaciais e Atmosféricas
IEEE	Institute of Electrical and Electronics Engineers
INPE	Instituto Nacional de Pesquisas Espaciais
ITU	International Telecommunication Union
JPA	Java Persistence API
LAC	Laboratório Associado de Computação e Matemática Aplicada
MEFs	Máquina de Estados Finitos
RF	Requisito funcional
RNF	Requisito não funcional
SDL	Specification and Description Language
SDL-GR	Specification and Description Graphic Representation
SDL-PR	Specification and Description Textual Phrase Representation
SID	Serviço de Informação e Documentação
UML	Unified Modeling Language

1 INTRODUÇÃO

O avanço intenso e progressivo da eletrônica e da computação, nas últimas décadas, tornou evidente um cenário, caracterizado pela presença de *software* de tipos e propósitos variados em todos os setores da sociedade. Como conseqüência disso, a sociedade passou a ser extremamente dependente da capacidade desses *softwares* de executar, correta e continuamente, as funções para as quais foram projetados e construídos. Em outras palavras, a garantia de qualidade e continuidade na execução, de praticamente todos os processos dos mais diversos setores da sociedade moderna, depende diretamente da qualidade do *software* empregado para automatizar as tarefas relativas às diversas atividades dos mesmos.

Entende-se por qualidade de *software*, o grau de conformidade que um produto de *software* apresenta em atender aos requisitos funcionais e não funcionais que tornaram necessário o desenvolvimento do mesmo. A qualidade de um produto de *software* está diretamente relacionada à qualidade do processo de *software* utilizado para a produção do mesmo (PRESSMAN, 2006).

São considerados sistemas críticos aqueles que, quando operam de forma inadequada (apresentando comportamento discrepante em relação ao que foi especificado), podem acarretar em perdas econômicas significativas, danos físicos ou ameaça a vidas humanas (SOMMERVILLE, 2003).

Sistemas de *software* para aplicação espacial, como aqueles embarcados em computadores de satélites, caso do INPE que tem como missão projetar e construir satélites para missões espaciais brasileiras, são classificados como sistemas críticos e, também complexos devido a forte interação com o *hardware* do computador, sensores, atuadores e outros dispositivos (SANTIAGO *et al*., 2006). Além disso, são de difícil substituição, em caso de falhas, devido ao aspecto não tripulado desse tipo de missão. Dessa forma, é

necessário um maior investimento para garantir a qualidade desse tipo de *software*.

Um dos recursos existentes dentro da Engenharia de *Software,* para a garantia da qualidade de *software,* é o processo de verificação e validação. O processo de verificação e validação tem como objetivo assegurar que o *software* cumpra com suas especificações e atenda às necessidades para o qual está sendo desenvolvido. Este processo deve estar ativo durante todo o ciclo de vida do *software*, desde as revisões de requisitos, continuando com as revisões de projeto, inspeções de código até chegar aos testes (SOMMERVILLE, 2003).

Verificação e validação são termos relacionados, porém, com significados distintos. As atividades de verificação têm o intuito de checar se o que foi desenvolvido funciona corretamente, respondendo a pergunta: “Estamos construindo o *software* corretamente?”. As atividades de validação têm por objetivo checar a conformidade do que foi desenvolvido em relação ao que foi especificado, respondendo a pergunta: ”Estamos construindo o *software* correto?” (BOEHM, 1979).

Para a realização do processo de verificação e validação, existem duas técnicas principais: inspeções de *software* e testes de *software* (SOMMERVILLE, 2003). As inspeções de *software* são técnicas estáticas, aplicadas durante todo o ciclo de vida do *software*, com o objetivo de analisar e verificar a conformidade dos artefatos produzidos aos padrões definidos para o projeto.

No contexto de processo de *software*, artefatos podem ser compreendidos como produtos resultantes das atividades de desenvolvimento. Exemplos de tipos de artefatos e alvos de inspeção são: documento de requisitos, diagramas, arquivos de código-fonte, etc.

Embora os termos falta, erro e falha sejam utilizados com freqüência, vários autores se referem a estes termos com definições distintas, como em

(BINDER, 2000) ou (SOMMERVILLE, 2003). No contexto desta dissertação serão adotadas as convenções abaixo usadas pelo grupo de testes do LAC (Laboratório Associado de Computação e Matemática Aplicada) e CEA (Ciências Espaciais e Atmosféricas) do INPE, as quais são adaptadas de (IEEE, 1990):

- Falta: Passo, processo ou definição de dado incorreto, por exemplo, uma instrução errada em uma rotina de *software*.

- Erro: Diferença entre um valor (computado, observado ou mensurado) ou uma condição e o que seria correto, segundo a especificação do *software*.

- Falha: Resultado incorreto percebido pelo usuário dado que o *software* está sendo executado.

Figura 1.1 – Relacionamento entre os conceitos: falta, erro e falha.

A Figura 1.1 apresenta o relacionamento entre os conceitos de falta, erro e falha. Em síntese, uma falta no *software* está no universo físico. Uma falta pode ou não produzir um erro (universo da informação), pois é possível que a rotina de *software* ou unidade de informação que contenha a falta nunca seja executada ou utilizada. Um erro pode ou não produzir uma falha (universo do usuário), pois o resultado da execução de um trecho de código que contenha

uma falta e consequentemente produza um erro pode ou não ser utilizado pelo *software* em execução.

A finalidade dos testes de *software* é detectar faltas latentes em um produto de *software* antes que o mesmo seja colocado em funcionamento. Trata-se de uma técnica dinâmica, que envolve a execução do *software* e a observação do seu comportamento e de suas saídas. Caso uma falha seja observada, o contexto da execução é armazenado e analisado para que seja possível encontrar e corrigir a falta que originou o erro que causou a falha em questão.

Um teste é considerado bem sucedido se consegue expor uma falha do sistema, fazendo com que o mesmo opere incorretamente, o que indica a existência de uma falta. É importante destacar que os testes devem ser capazes de indicar a presença e não a ausência de faltas em *software*.

Os testes funcionais (conhecidos como testes de caixa preta) consistem em uma abordagem na qual os testes são derivados da especificação do *software*. A funcionalidade, foco do teste, é encarada como uma “caixa-preta” cujo comportamento só pode ser avaliado por meio da comparação das saídas obtidas em relação às saídas desejadas para entradas conhecidas (SOMMERVILLE, 2003).

Testes estruturais (conhecidos como testes de caixa branca) são derivados do conhecimento da estrutura e da codificação do programa. De maneira geral, esse tipo de teste é aplicado a unidades de programa relativamente pequenas, como métodos ou funções, pois seria muito complicado planejar testes baseados em código-fonte sob a perspectiva do sistema completo. O testador nesse caso utiliza os conhecimentos sobre a estrutura da codificação para derivar os dados para o teste (SOMMERVILLE, 2003).

Um caso de teste pode ser definido como um conjunto de entradas, condições de execução e resultados esperados, desenvolvido com o objetivo de exercitar uma porção particular de *software*, verificando sua conformidade em relação à especificação de requisitos (IEEE, 1990).

É conhecido o fato de que a realização exaustiva de testes de *software* em sistemas complexos, como os de aplicação espacial, por exemplo, é impraticável, pois as atividades de testes consumiriam muito tempo, devido à complexidade desse tipo de *software* (SANTIAGO *et al*., 2006). Dessa forma, torna-se interessante a utilização de técnicas que permitam a geração automática de casos de teste a partir de especificações formais que possam ser manipuladas por computador. Além disso, é comum a necessidade de utilizar algum critério de cobertura que permita a redução da quantidade de casos de testes gerados a um número que seja possível de ser executado, tentando manter o mais elevado possível à possibilidade de encontrar falhas por meio do conjunto de casos de testes gerados (LEE *et al*., 1996), (MARTINS *et al*., 2000), (MYERS, 1979) e (PIMONT *et al*.1979).

Sistemas de *software* de aplicação espacial são considerados sistemas reativos, pois podem ser representados como um conjunto de estados e transições entre estados, que ocorrem por meio de estímulos também conhecidos como eventos.

Máquinas de Estados Finitos (MEFs) são uma técnica formal de descrição, utilizada para a representação desse tipo de sistema, devido ao fato dessa técnica lidar naturalmente com estados e transições. Mesmo assim, é complicado representar características comuns dos sistemas complexos modernos, tais como hierarquia e paralelismo utilizando MEFs. Por esse motivo torna-se interessante o uso de técnicas de mais alto nível que ofereçam essas facilidades e possuam também o formalismo necessário para possibilitar a manipulação de especificações de sistemas reativos pelo computador, com o

objetivo de geração de casos de teste. Entre essas técnicas de mais alto nível, pode-se citar: *Statecharts* (HAREL, 1987), (AMARAL, 2005), (SANTIAGO *et al.*, 2006), Redes de Petri (DESEL *et al.*, 2007), SDL (WONG, 2003), entre outras.

Foram realizados estudos explorando a utilização de *Statecharts* como técnica formal de descrição para *software* de aplicação espacial do INPE, visando a derivação de casos de teste (de caixa preta), como em (AMARAL, 2005), (SANTIAGO *et al*., 2006), (ARANTES *et al*., 2008), (SOUZA *et al*., 2008) e (SANTIAGO *et al*., 2008).

A linguagem SDL (*Specification and Description Language*) é muito utilizada para a descrição de sistemas de *software* no setor de telecomunicações, que em grande parte são considerados sistemas reativos. Devido as similaridades entre as aplicações espaciais a as aplicações de telecomunicações, julgou-se interessante explorar a utilização da SDL para a especificação e geração automática de casos de teste.

Este trabalho possui dois objetivos principais. O primeiro é explorar a simulação de comportamento, em termos de mudança de estados, de especificações em SDL, utilizando um *framework* desenvolvido para este propósito, denominado J-SDL. A idéia é utilizar os recursos de simulação oferecidos pelo J-SDL para validar especificações formais, se tais especificações foram coerentemente elaboradas.

O segundo é explorar a geração automática de casos de teste (de caixa preta), utilizando critérios de cobertura, a partir da simulação de especificações em SDL com a finalidade de detecção de falhas no *software*.

2 SDL

A SDL (*Specification and Description Language)* é uma linguagem formal para a especificação e descrição de sistemas complexos, que são reativos, mantida pela ITU (*International Telecommunication Union*) sob a denominação de Recomendação Z.100. Sistemas reativos são aqueles que respondem a algum estímulo também conhecido como sinal ou evento. Esta linguagem possibilita a representação formal de sistemas quanto aos aspectos de comportamento, estrutura, comunicação e descrição de dados.

A primeira versão da Recomendação Z.100 foi disponibilizada pela CCITT (*Comitê Consultatif International Télégraphique et Téléphonique*, atual ITU) em 1976 apresentando apenas uma abordagem rudimentar de representação de comportamento. Em 1984 foi lançada uma nova versão da linguagem, abordando também representação de estrutura e de comunicação. Apenas em 1988, a SDL recebeu embasamento formal e atingiu o status de técnica formal de descrição. Os primeiros conceitos de orientação a objetos foram incluídos na linguagem em 1992 (ELLSBERGER *et al*., 1997). A versão atual da SDL foi disponibilizada pela ITU em 2002, apresentando suporte aprimorado para modelagem orientada a objetos e recursos adicionais para geração de código-fonte.

No escopo da SDL, os termos especificação e descrição possuem significados distintos. O termo especificação é utilizado para representar o comportamento esperado de um sistema, enquanto o termo descrição é utilizado para representar o comportamento real de um sistema. Como a forma de utilização da SDL para especificar ou para descrever é a mesma, é comum o uso do termo especificação de forma genérica para referenciar tanto especificações quanto descrições (ITU, 2002).

2.1 Conceitos básicos

Uma especificação em SDL é um modelo formal que define as propriedades relevantes de um sistema existente ou a ser construído ou implementado. No paradigma da SDL o mundo é divido em duas partes: o sistema especificado e o ambiente, onde o ambiente é tudo que circunda e se comunica com o sistema em questão.

A comunicação entre ambiente e sistema ocorre através da troca de sinais. Desta forma, se existe no ambiente um evento relevante para o sistema, este evento é representado na especificação por meio de um sinal que pode ser enviado pelo ambiente para o sistema. A troca de sinais entre ambiente e sistema ocorre de forma bidirecional conforme ilustra a Figura 2.1.

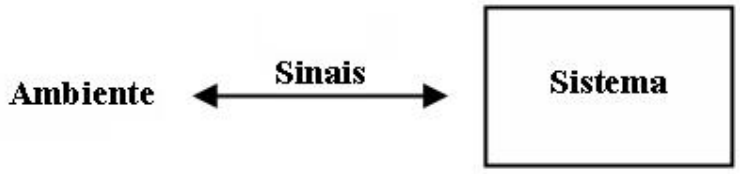

Figura 2.1 – Interação entre ambiente e sistema.

Uma especificação, quando apropriado, pode ser realizada de forma hierárquica através do particionamento sucessivo de um sistema em níveis diferentes de abstração. Cada nível definido na especificação pode possuir a sua própria especificação, ou seja, pode continuar a ser decomposto em sub-níveis e assim sucessivamente até que sejam representados todos os aspectos desejados do sistema em questão, conforme apresentado na Figura 2.2.

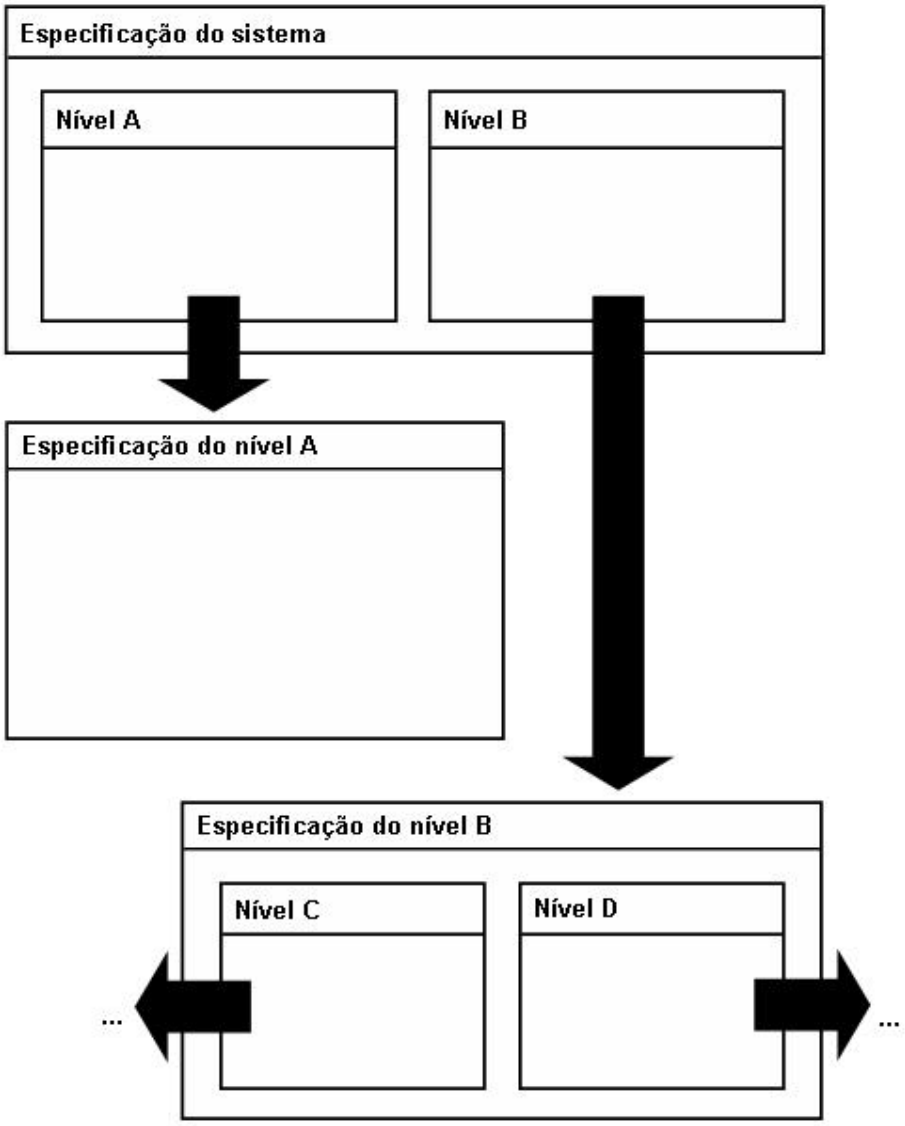

Figura 2.2 – Particionamento em níveis de abstração.

Para possibilitar que um nível de abstração seja reutilizado dentro de uma ou mais especificações, a SDL adota os conceitos de tipo e instância. Cada nível de abstração identificado durante o fracionamento pode ser definido como um tipo de nível e ser instanciado na definição de outros níveis ou tipos de níveis. Através do uso de tipos, é possível também instanciar conjuntos de instâncias,

similar a um vetor. A Figura 2.3 ilustra a utilização de tipos, instâncias e conjuntos.

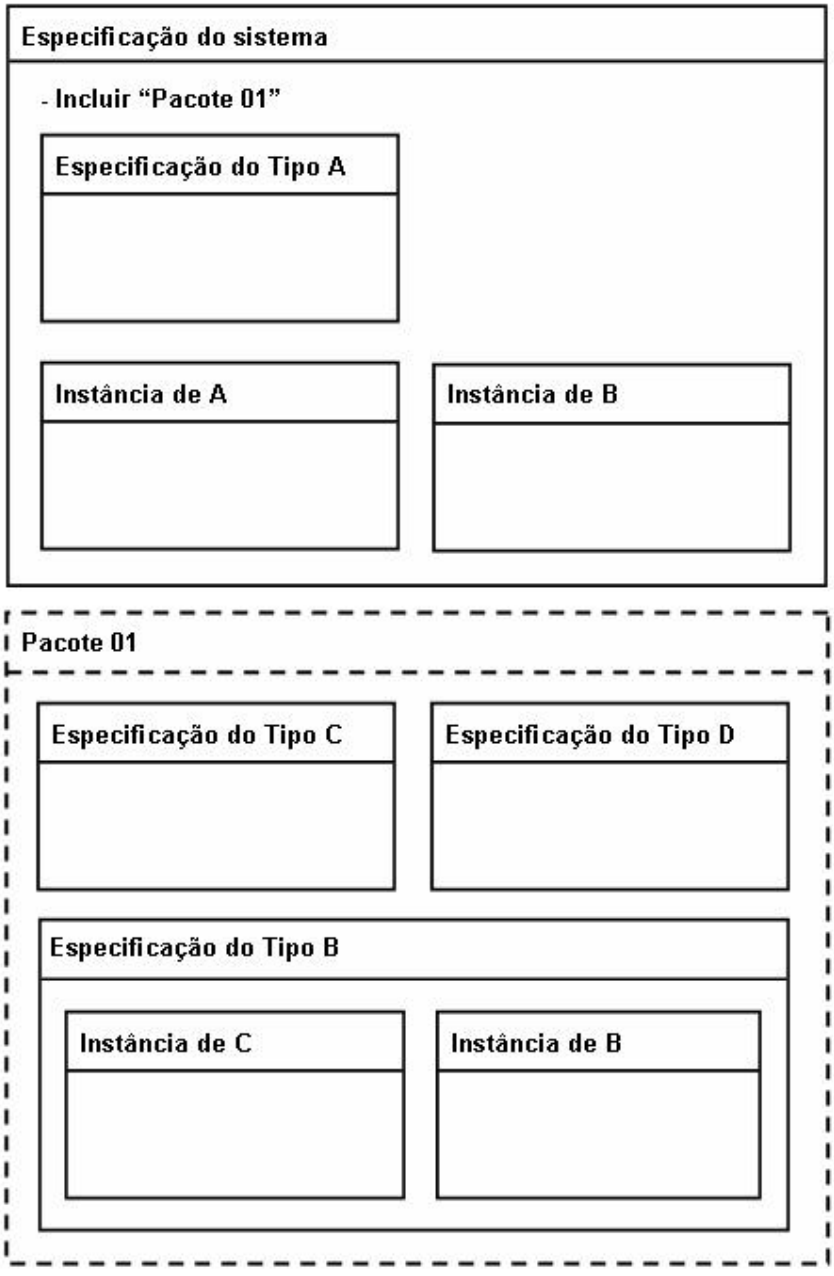

Figura 2.3 – Utilização de tipos, instâncias e conjuntos.

É opcional, porém recomendada, a organização das especificações de tipos de níveis em pacotes para simplificar a manipulação de níveis de abstração afins

entre si. Um pacote é um agrupador que permite que um conjunto de especificações seja importado em outras especificações.

2.2 Elementos Principais

A SDL oferece duas formas de representação uma textual (SDL-PR) e uma gráfica (SDL-GR). Ambas oferecem formas equivalentes de representação dos mesmos elementos. Os elementos da SDL podem ser organizados em três grupos: elementos de estrutura, elementos de comunicação e elementos de comportamento.

2.2.1 Elementos de estrutura

Os elementos de estrutura são aqueles que permitem a representação dos três níveis de abstração existentes no paradigma da SDL: nível de sistema, nível de bloco e nível de processo.

Toda especificação possui obrigatoriamente um, e apenas um, nível de sistema. Trata-se do nível mais alto de abstração da especificação, onde são definidos os aspectos de comunicação entre sistema e ambiente. Pode-se dizer que este nível é a raiz da especificação.

O elemento da SDL que permite a representação do nível de sistema é o elemento *System*. A Figura 2.4 apresenta as formas gráfica e textual de representação do nível de sistema.

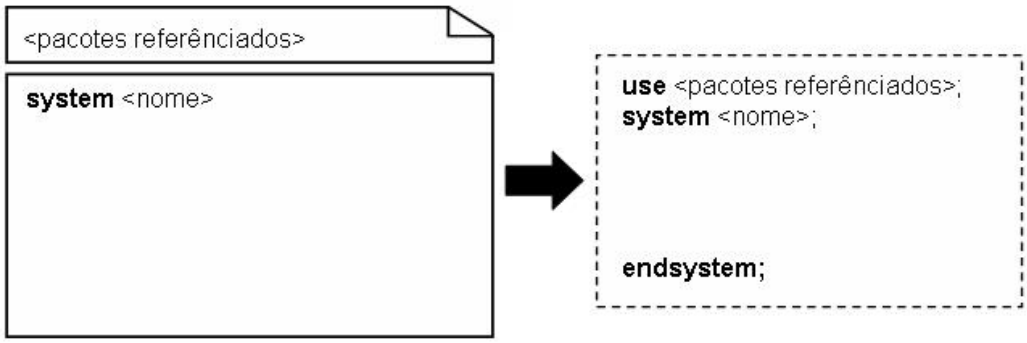

Figura 2.4 – Definição de níveis de sistema.

O nível de bloco funciona como um *container* para outros níveis, sejam eles de bloco ou de processo. A utilização desse tipo de nível não é obrigatória, porém se torna muito útil em projetos reais, onde os diagramas tendem a ficar muito grandes e complexos. A quebra de um nível de sistema ou de bloco em níveis de blocos menores facilita significativamente a manipulação dos diagramas, além de possibilitar que essas "partes" da especificação sejam reutilizadas em outros níveis de abstração, até mesmo de outros sistemas.

O elemento que permite a declaração de tipos de blocos é o *Block Type*. A Figura 2.5 apresenta as formas de representação deste elemento.

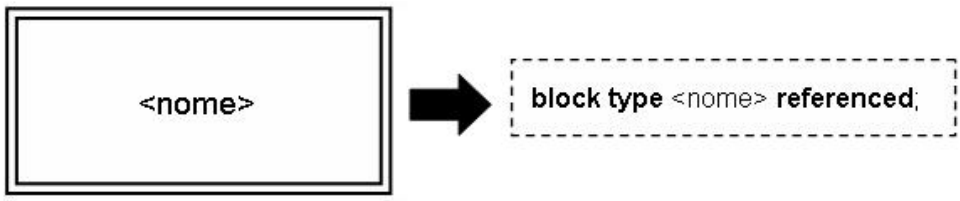

Figura 2.5 – Declaração de tipos de bloco.

Um tipo de bloco pode ser declarado dentro da definição de um pacote, de um sistema ou de outro tipo de bloco. A definição do nível de abstração representado pelo tipo de bloco é realizada separadamente, conforme apresenta a Figura 2.6.

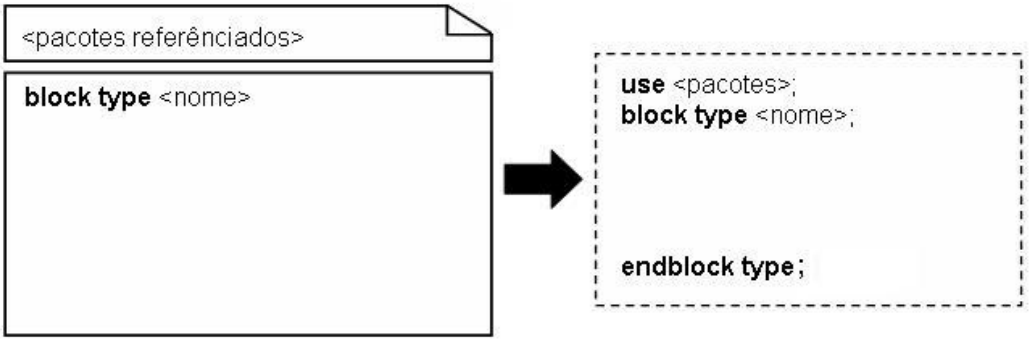

Figura 2.6 – Definição de níveis de bloco.

A partir de um tipo de bloco é possível instanciar blocos nas definições de níveis de sistemas ou de outros níveis de blocos. A Figura 2.7 apresenta as formas de representação de instâncias de blocos e de conjuntos de instâncias de blocos respectivamente.

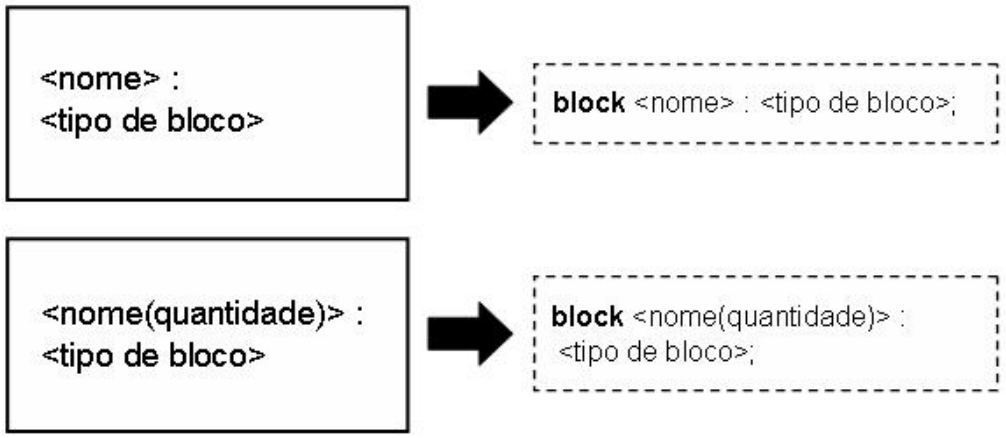

Figura 2.7 – Instância de blocos e conjuntos de blocos.

A Figura 2.8 ilustra o uso de um bloco b1 do tipo blocoPrincipal na definição do sistema exemplo.

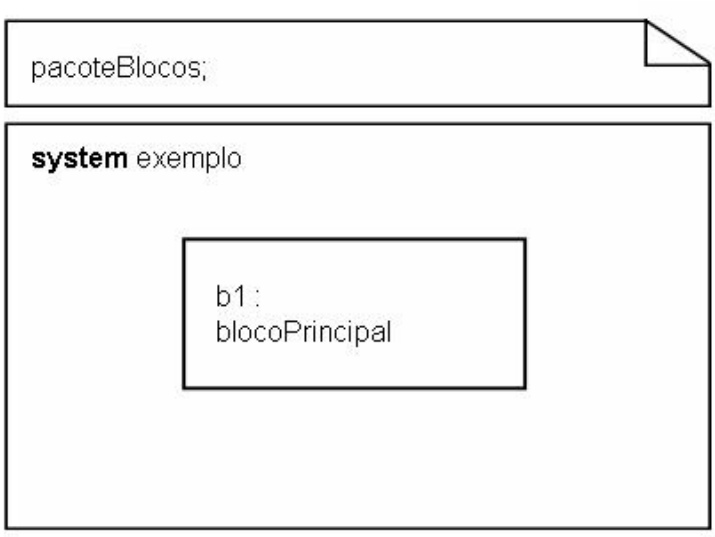

Figura 2.8 – Definição de sistema através de instâncias de bloco.

O nível de processo permite a representação de partes de comportamento do sistema, por meio do encapsulamento de máquinas de estados finitos estendidas que comunicam entre si através da troca de sinais.

O elemento que permite a declaração de tipos de processo é o *Process Type*. A Figura 2.9 apresenta as formas de representação deste elemento.

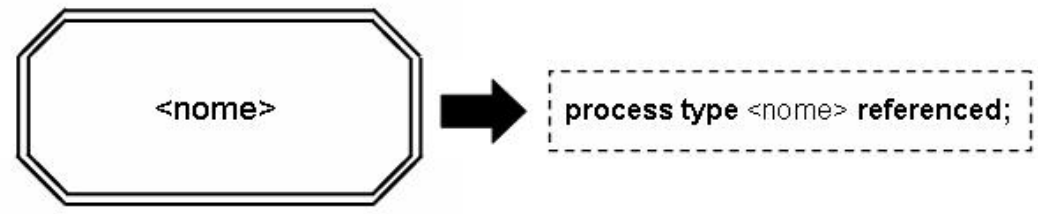

Figura 2.9 – Declaração de tipos de processo.

Um tipo de processo pode ser declarado dentro da definição de um pacote, de um sistema ou de um bloco. A definição do nível de abstração representado pelo tipo de processo é realizada separadamente, conforme apresenta a Figura 2.10.

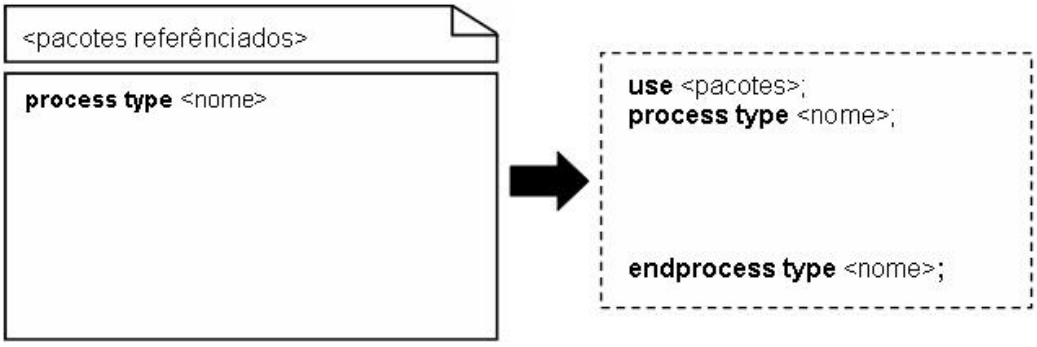

Figura 2.10 – Declaração de tipos de bloco.

A partir de um tipo de processo é possível instanciar processos nas definições de níveis de sistemas ou de níveis de blocos. A Figura 2.11 apresenta as formas de representação de instâncias de processos e de conjuntos de instâncias de processos respectivamente.

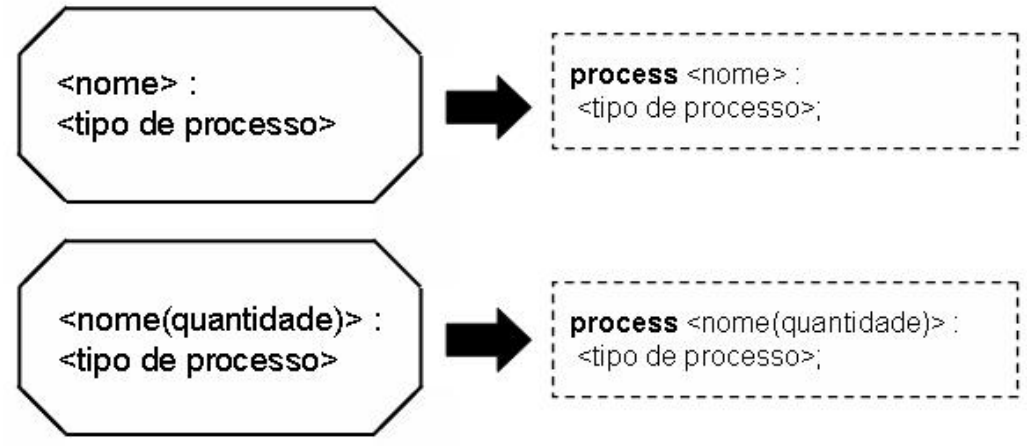

Figura 2.11 – Instância de processos e conjuntos de processos.

A Figura 2.12 ilustra o uso de um processo p1 do tipo processo1 na definição do tipo de bloco exemplo.

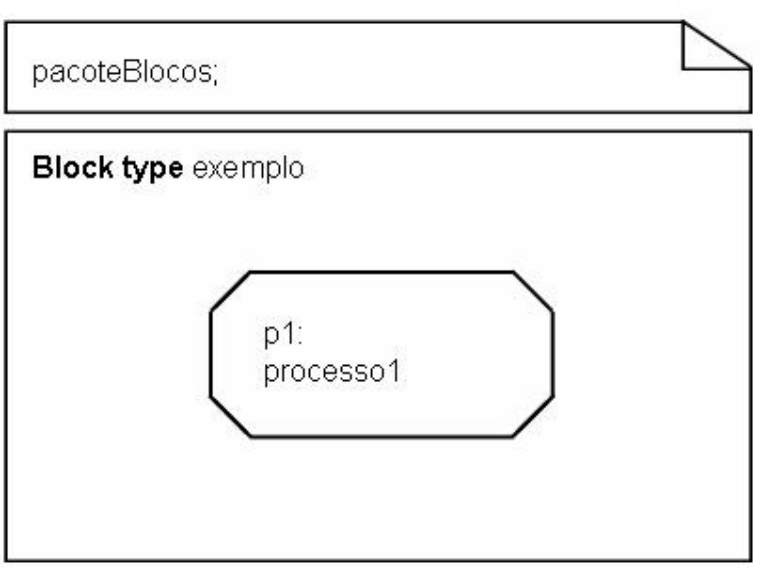

Figura 2.12 – Definição de tipo de bloco através de instâncias de processo.

2.2.2 Elementos de comunicação

O principal mecanismo de comunicação utilizado dentro da SDL é a troca assíncrona de sinais. É através do envio de sinais (parametrizados ou não) que os processos trocam informações e se mantêm sincronizados. É importante destacar que os sinais podem ser enviados e recebidos por processos ou pelo ambiente.

Para que um sinal seja transmitido é necessário um meio de transmissão. Na SDL este meio é denominado canal. Os canais são utilizados para definir uma rota de comunicação entre dois elementos de estrutura. O elemento da SDL que permite a definição de um canal é o *Channel*. A Figura 2.13 apresenta as formas gráfica e textual de representação desse elemento.

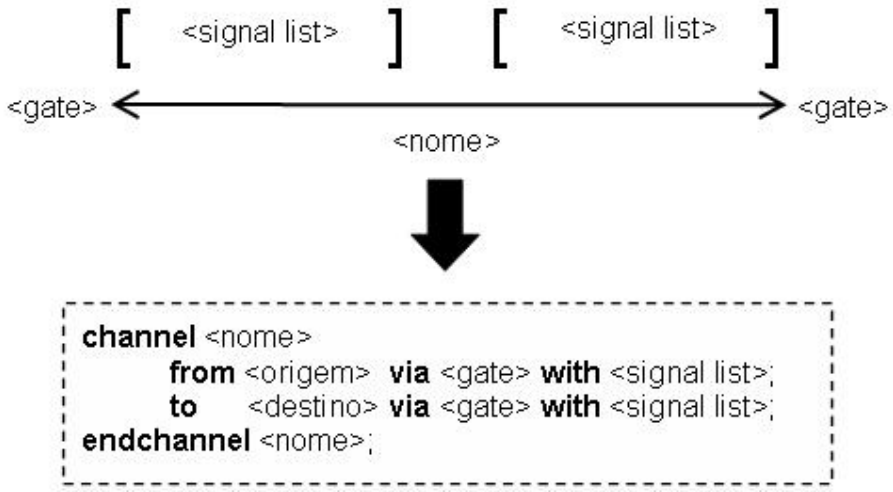

Figura 2.13 – Definição de canais.

Para definir um canal é preciso que o elemento de estrutura possua algum ponto de conexão. O elemento que representa este ponto de conexão é chamado *Gate*. Elementos do tipo gate podem ser criados na definição dos elementos: *Block Type, Process Type, System*.

Em SDL, os sinais são emitidos por *broadcast*, por isso é necessário delimitar por onde cada sinal deve ou não propagar. Conforme ilustrado pela Figura 2.13, o elemento *Channel* possui duas listas de sinais associadas aos seus dois *Gates*. Essas listas de canais funcionam como filtros que determinam por onde um sinal pode propagar. Caso um sinal não esteja na lista de sinais de um *Channel*, o mesmo não será propagado.

A Figura 2.14 ilustra o uso de um canal para conectar um processo a um bloco.

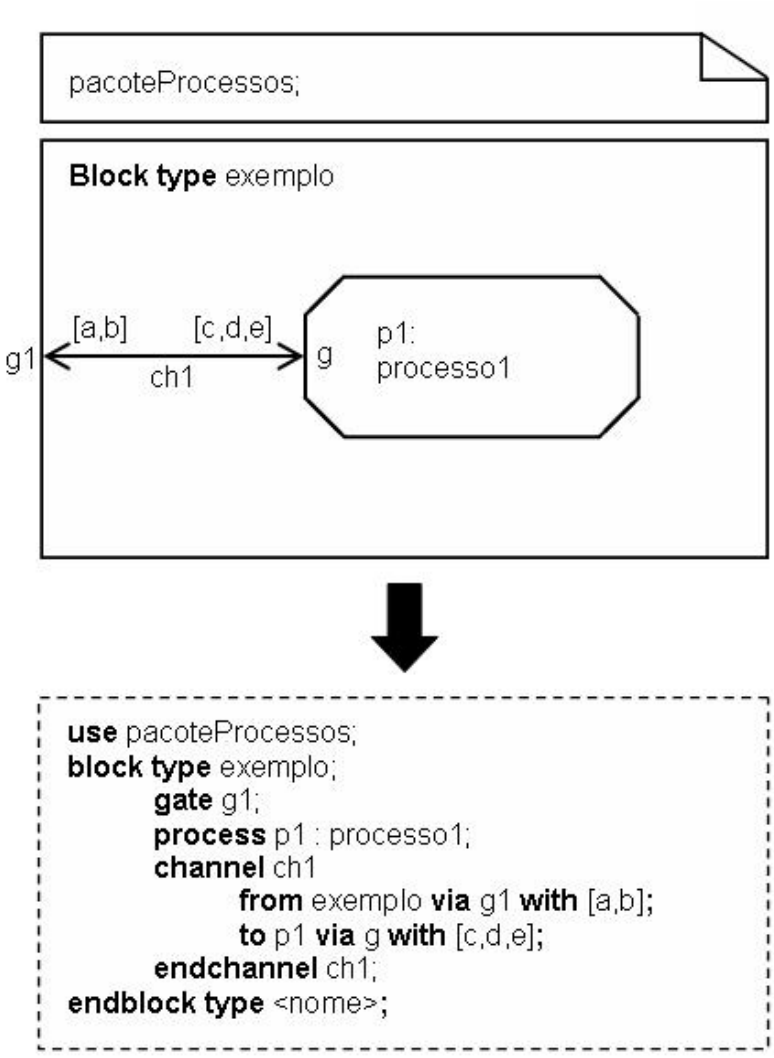

Figura 2.14 – Exemplo de utilização de canais.

2.2.3 Elementos de comportamento

O comportamento dinâmico em SDL é definido dentro dos processos. Uma instância de processo é uma maquina de estados estendida que se comunica com outras instâncias de processos ou com o ambiente, por meio da troca assíncrona de sinais. É possível a existência de mais de uma instância do mesmo processo. Cada instância de processo possui um identificador único, o que possibilita o envio de sinais diretamente à determinada instância de um processo.

Diversos elementos são utilizados para descrever o comportamento de tipos de processos em termos de máquinas de estados estendida. Entre eles podemos destacar os elementos: *State, Start, Input, Output, Save e Decision.*

O elemento *State* representa um estado dentro de um tipo de processo. A Figura 2.15 apresenta suas formas de representação.

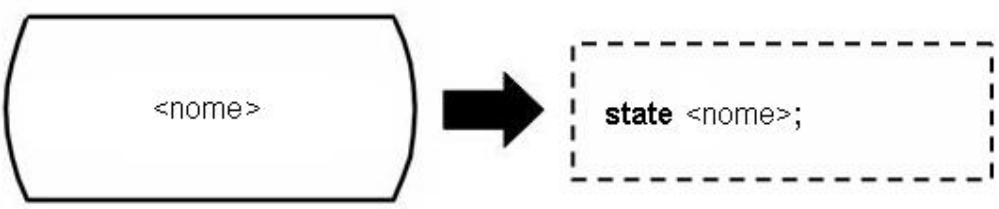

Figura 2.15 – Representação do elemento *State.*

O elemento *Start* determina qual o estado inicial do tipo de processo. A Figura 2.16 apresenta suas formas de representação.

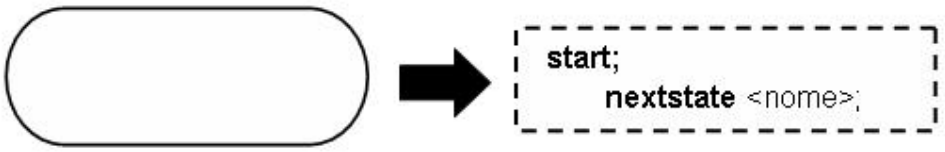

Figura 2.16 – Representação do elemento *Start.*

O elemento *Input* é utilizado para especificar a sensibilidade de um estado a um sinal e disparar uma transição para outro estado quando a instância do processo receber determinado sinal. A Figura 2.17 apresenta suas formas de representação.

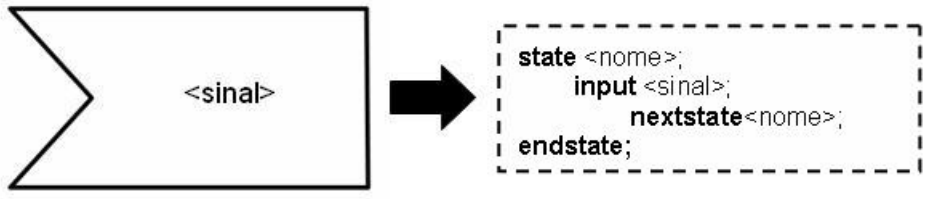

Figura 2.17 – Representação do elemento *Input*.

O elemento *Output* é utilizado para especificar o envio de um sinal durante a transição de um estado para outro. A Figura 2.18 apresenta suas formas de representação.

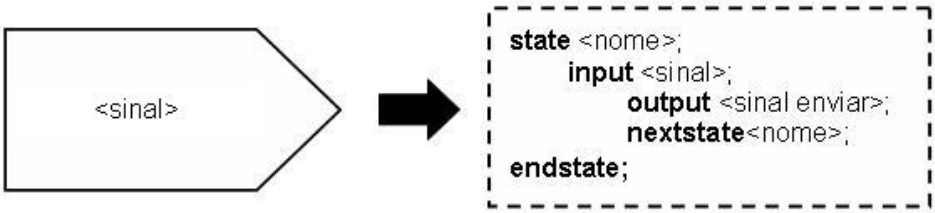

Figura 2.18 – Representação do elemento *Output.*

Em algumas situações pode ser necessário armazenar um sinal recebido para que o mesmo seja consumido em outro momento. O elemento *Save* possibilita que um sinal recebido seja armazenado e disparado novamente quando acontecer à próxima transição de estado dentro da instância do processo. A Figura 2.19 apresenta as formas de representação do elemento *Save*.

É possível definir elementos *Input* ou *Save* genéricos que sejam sensíveis a qualquer sinal recebido. Para isso basta identificar o sinal com um asterisco (*).

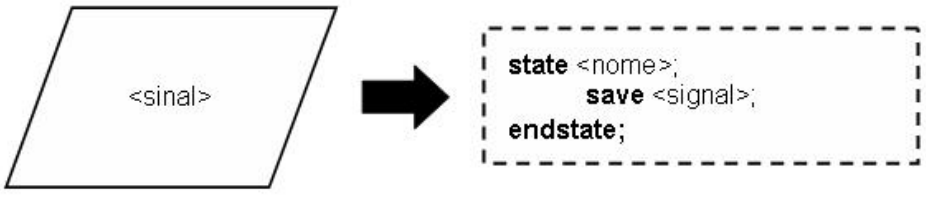

Figura 2.19 – Representação do elemento *Save*.

Uma transição pode possuir um desvio que dependa de alguma condição, o elemento *Decision* é utilizado para representar este desvio. A Figura 2.20 apresenta suas formas de representação.

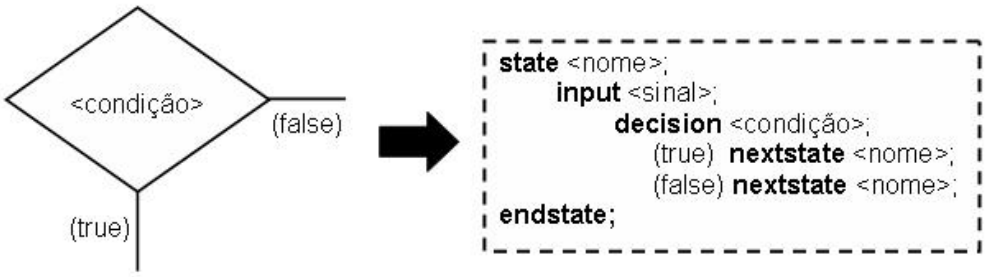

Figura 2.20 – Representação do elemento *Decision*.

A Figura 2.21 apresenta um exemplo de definição de tipo de processo utilizando alguns dos elementos de comportamento apresentados.

O tipo de processo p1 recebe e envia sinais pelo *Gate* g1. Ao ser instanciada uma instância desse de p1, automaticamente é realizada a transição para o estado inicial eA. O estado eA é sensível ao sinal sb. Quando uma instância de p1 se encontra no estado eA e recebe o sinal sb, ocorre a transição do estado eA para o estado eB. O estado eB por sua vez é sensível ao sinal sa. Quando uma instância de p1 recebe o sinal sA, ocorre a transição do estado eB para eA.

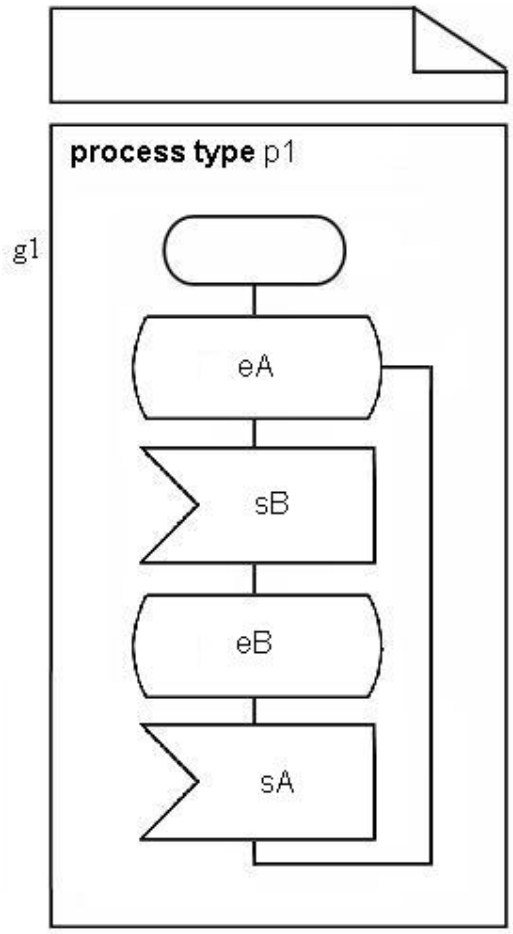

Figura 2.21 – Exemplo de definição de comportamento.

A SDL conta com uma série de outros elementos para a especificação de comportamento. O documento de definição da especificação Z.100 apresenta todos os elementos de forma detalhada.

3 O FRAMEWORK J-SDL

Um *framework*, dentro do paradigma da programação orientada a objetos, pode ser compreendido como uma coleção de classes, interfaces e padrões dedicados a resolver um conjunto de problemas relacionados, por meio de uma arquitetura de *software* flexível e extensível. Um exemplo bastante conhecido de *framework* é o JPA (*Java Persistence API*) (BISWAS *et al.,* 2008) que oferece um conjunto de classes e interfaces em Java para resolver o problema de mapeamento e persistência de objetos em bancos de dados relacionais.

O desenvolvimento de um *framework* é justificável quando o mesmo possui potencial para ser utilizado por vários sistemas de *software*, pois a complexidade de projetar e codificar uma solução genérica é alta, o que torna o processo de desenvolvimento de um *framework* bastante dispendioso para atender a uma única instância de problema.

O J-SDL é um *framework* Java, desenvolvido com o objetivo de possibilitar a simulação de comportamento de sistemas reativos a partir de especificações em SDL, possibilitando por meio do estímulo de eventos conhecidos como sinais, observar a reação da especificação em termos de mudança de estados.

A decisão de construir o J-SDL no formato de um *framework* é justificada pelo potencial de utilização de seus recursos de simulação, para o desenvolvimento de diversas ferramentas computacionais destinadas a auxiliar as atividades do processo de verificação e validação, como por exemplo, aplicações que utilizem a simulação de especificação para derivar casos de teste, segundo algum critério de cobertura.

A seguir serão apresentados os principais aspectos do J-SDL, segundo as visões de análise, projeto e implantação.

3.1 Visão de Análise

Requisitos podem ser definidos como características ou restrições a serem atendidas por um sistema de *software* (SOMMERVILLE, 2003). Este tópico tem como objetivo documentar os requisitos funcionais e não funcionais identificados para o desenvolvimento do J-SDL.

Os requisitos funcionais representam as funcionalidades que um *software* possui ou deve possuir (SOMMERVILLE, 2003). A Tabela 3.1 apresenta todos os requisitos funcionais identificados para o J-SDL, sendo cada um deles detalhados posteriormente. Foi adotado como padrão, utilizar o prefixo RF para compor o início do identificador de cada requisito funcional.

Tabela 3.1 – Requisitos funcionais do J-SDL.

ID	DESCRIÇÃO
RF01	Suporte a elementos básicos de estrutura.
RF02	Suporte a elementos de elementos básicos de comunicação.
RF03	Suporte a elementos de elementos básicos de processo.
RF04	Consultar o estado atual da simulação.
RF05	Consultar o contador de ativações por elemento *Input*.
RF06	Consultar o contador de ativações por elemento *State*.
RF07	Estimular sinais.
RF08	Reiniciar a simulação.

Os requisitos não funcionais representam restrições sobre as funcionalidades de um *software* (SOMMERVILLE, 2003). A Tabela 3.2 apresenta todos os requisitos não funcionais identificados para o J-SDL, sendo cada um deles detalhados posteriormente. Foi adotado como padrão, utilizar o prefixo RNF para compor o início do identificador de cada requisito não funcional.

Tabela 3.2 – Requisitos não funcionais do J-SDL.

ID	DESCRIÇÃO
RNF01	Independência de plataforma.
RNF02	Independência de software comercial.
RNF03	Facilidades para expansão e adaptação.

3.1.1 RF01 - Suporte a elementos básicos de estrutura

Em SDL, são considerados elementos básicos de estrutura: *System*, *Block* e *Process.* Para simular o comportamento de especificações em SDL é necessário definir estruturas de dados que permitam a manipulação da especificação em memória. Além disso, é necessário definir também regras para propagação de sinais que permitam que a especificação em memória reaja por meio de transições de estado.

Os elementos *Block* e *System* são muito parecidos, possuindo os mesmos atributos e as mesmas regras para propagação de sinais, mesmo assim, é necessária a distinção dos dois elementos, em termos de estrutura de dados, para manter conformidade com a especificação oficial da SDL (ITU, 2002). As estruturas de dados que mapeiam esses dois elementos devem possuir os atributos: *Name* e *Gates*. A Tabela 3.3 apresenta a finalidade de cada um desses atributos.

Tabela 3.3 – Atributos dos elementos *System* e *Block*.

Atributo	Finalidade
Name	Permitir a identificação única da instância de elemento de estrutura em questão.
Gates	Manter referência às instâncias do elemento *Gate* utilizadas como pontos de conexão para elementos *Channel* relativos à instância de elemento de estrutura em questão.

O comportamento esperado para as instâncias de elementos *System* e *Block* em relação ao envio e recebimento de sinais é apresentado pela Figura 3.1, por meio de um diagrama de atividades da UML (*Unified Modeling Language*).

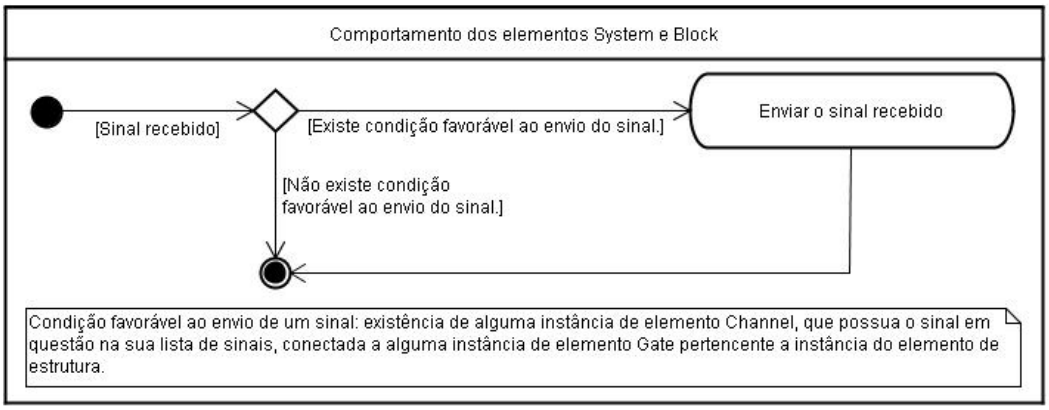

Figura 3.1 – Comportamento dos elementos *System* e *Block.*

A estrutura de dados que mapeia o elemento *Process* deve possuir os atributos: *Name*, *Gates*, *States*, *Signals in buffer*, *Current state* e *Default state*. A Tabela 3.4 apresenta a finalidade de cada um desses atributos.

Tabela 3.4 – Atributos do elemento *Process*.

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Process*.
Gates	Manter referência às instâncias do elemento *Gate* utilizadas como pontos de conexão para elementos *Channel* relativos à instância de elemento *Process*.
States	Manter referência às instâncias do elemento *State* utilizadas para definir as possibilidades de estado da instância de elemento *Process*.
Signals in buffer	Manter referência às instâncias do elemento *Signal* que estejam em *buffer* para a instância de elemento *Process*.

Current state	Manter referência à instância do elemento *State* que represente o estado atual da instância de elemento *Process*.
Default state	Manter referência à instância do elemento *State* que represente o estado padrão da instância de elemento *Process*.

O comportamento esperado para o elemento *Process* em relação ao envio e recebimento de sinais é apresentado pela Figura 3.2, por meio de um diagrama de atividades.

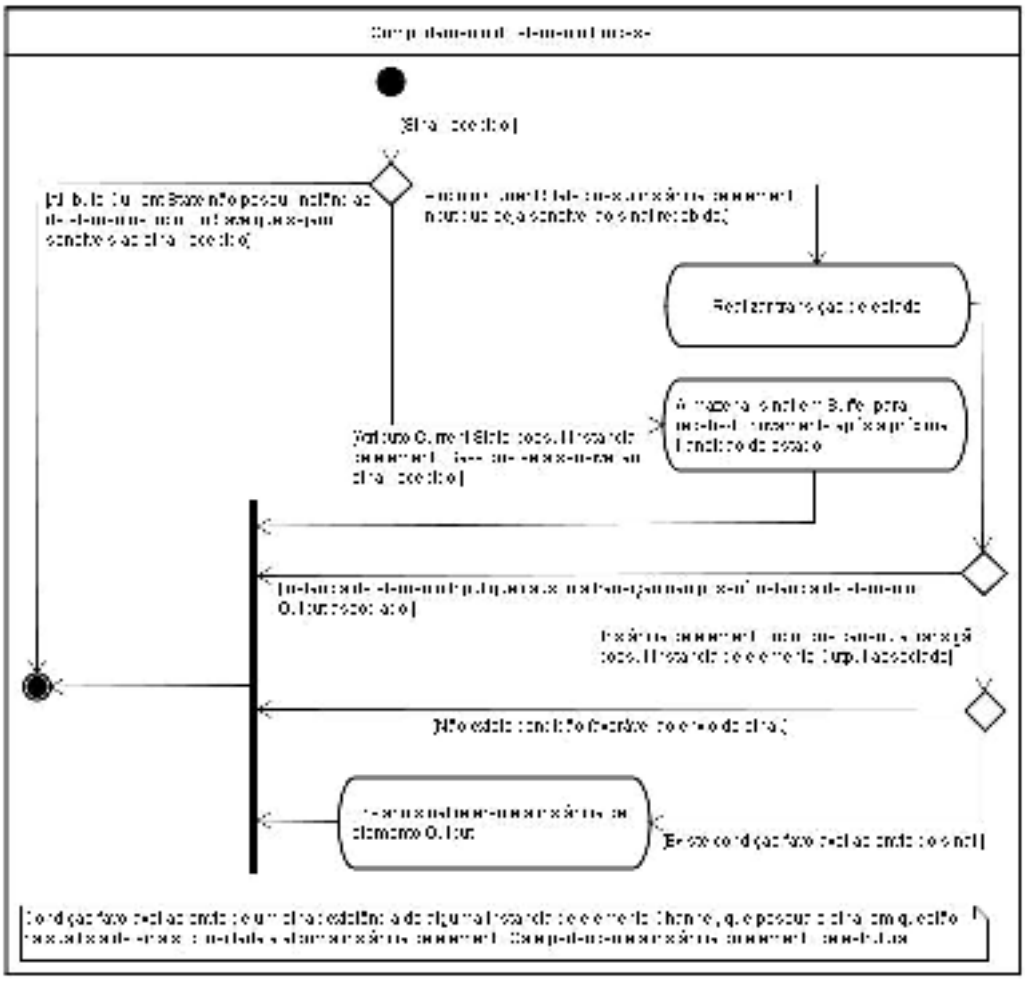

Figura 3.2 – Comportamento do elemento *Process.*

3.1.2 RF02 - Suporte a elementos básicos de comunicação

Os elementos básicos de comunicação utilizados em SDL são: *Gate*, *Channel* e *Signal*. É importante destacar que esses elementos possuem a função de representar as permissões de propagação de estímulos (sinais), e não de realizar propagação. Os elementos que efetivamente realizam a propagação de sinais (durante a simulação) são os elementos de estrutura. Dessa forma os elementos de comunicação não possuem comportamento para o recebimento e envio de sinais.

A estrutura de dados que mapeia o elemento *Gate* deve possuir os atributos: *Name*, *Channels*, *Owner*. A Tabela 3.5 apresenta a finalidade de cada um desses atributos.

Tabela 3.5 – Atributos do elemento *Gate*.

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Gate*.
*Channel*s	Manter referência às instâncias do elemento *Channel que* utilizam a instância de elemento *Gate* como ponto de conexão.
Owner	Manter referência à instância do elemento de estrutura "proprietária" da instância de elemento *Gate*.

A estrutura de dados que mapeia o elemento *Channel* deve possuir os atributos: *Name*, *Gate A*, *Gate B*, *Signal List A* e *Signal List B*. A Tabela 3.6 apresenta a finalidade de cada um desses atributos.

Tabela 3.6 – Atributos do elemento *Channel.*

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Channel*.

Gate A	Manter referência à instância de elemento *Gate* utilizadas como ponto de conexão para a extremidade A da instância do elemento *Channel*.
Gate B	Manter referência à instância de elemento *Gate* utilizadas como ponto de conexão para a extremidade B da instância do elemento *Channel*.
Signal List A	Manter referência aos elementos *Signal* utilizados para especificar quais sinais podem ser propagados no sentido do Elemento Gate A, por meio da instância de elemento *Channel*.
Signal List B	Manter referência aos elementos *Signal* utilizados para especificar quais sinais podem ser propagados no sentido do Elemento Gate B, por meio da instância de elemento *Channel*.

A estrutura de dados que mapeia o elemento *Signal* deve possuir o atributo: *Name*. A Tabela 3.7 apresenta a finalidade desse atributo.

Tabela 3.7 – Atributos do elemento *Signal.*

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Signal*.

3.1.3 RF03 - Suporte a elementos básicos de comportamento

Os elementos básicos de comportamento utilizados em SDL são: *State*, *Input, Output* e *Save*. Esses elementos possuem a finalidade de descrever o comportamento, em termos de mudança de estados, de instâncias de elementos *Process* ao receber e enviar sinais. Os elementos que efetivamente executam o comportamento descrito pelos elementos de comportamento (durante a simulação) são os elementos de estrutura.

A estrutura de dados que mapeia o elemento *State* deve possuir os atributos: *Name*, *Owner*, *Inputs* e *Saves*. A Tabela 3.8 apresenta a finalidade de cada um desses atributos.

Tabela 3.8 – Atributos do elemento *State.*

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *State*.
Owner	Manter referência à instância do elemento *Process* "proprietária" da instância de elemento *State*.
Inputs	Manter referência às instâncias de elemento *Input associadas à instância de elemento State.*
Saves	Manter referência às instâncias de elemento *Save associadas à instância de elemento State.*

A estrutura de dados que mapeia o elemento *Input* deve possuir os atributos: *Name*, *Signal*, *From State*, *To State*, *Output*. A Tabela 3.9 apresenta a finalidade de cada um desses atributos.

Tabela 3.9 – Atributos do elemento *Input.*

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Input*.
Signal	Manter referência à instância do elemento *Signal* associada à instância de elemento *Input*.
From State	Manter referência à instância do elemento *State relativa ao estado de origem da transição associada* à instância de elemento *Input*.
To State	Manter referência à instância do elemento *State relativa ao estado de destino da transição associada* à instância de elemento *Input*.
Output	Manter referência à instância do elemento *Output que*

	pode estar associada à instância de elemento *Input*.

A estrutura de dados que mapeia o elemento *Output* deve possuir os atributos: *Name*, *Signal* e *Input*. Tabela 3.10 apresenta a finalidade de cada um desses atributos.

Tabela 3.10 – Atributos do elemento *Output.*

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Output*.
Signal	Manter referência à instância do elemento *Signal* associada à instância de elemento *Output*.
Input	Manter referência à instância do elemento *Input que pode estar associada* à instância de elemento *Output*.

A estrutura de dados que mapeia o elemento *Save* deve possuir os atributos: *Name*, *Signal*, *Owner*. Tabela 3.11 apresenta a finalidade de cada um desses atributos.

Tabela 3.11 – Atributos do elemento *Save..*

Atributo	**Finalidade**
Name	Permitir a identificação única da instância de elemento *Save*.
Signal	Manter referência à instância do elemento *Signal* associada à instância de elemento *Save*.
Owner	Manter referência à instância do elemento *State* "proprietário" da instância de elemento *Save*.

3.1.4 RF04 - Consultar o estado atual da simulação

É necessário que o J-SDL permita consultar o estado atual de uma especificação durante a simulação de comportamento. Entende-se por estado

atual da simulação o nome de todas as instâncias de elemento *State* que sejam referenciadas por alguma instância de elemento Process como elemento State atual.

3.1.5 RF05 - Consultar número de ativações por elemento Input

É necessário que o J-SDL permita consultar o contador de ativações ocorridas por instância de elemento Input desde o início da simulação. Esta informação pode ser útil para o desenvolvimento de aplicativos que utilizem critérios de cobertura baseados em elementos *Input*.

3.1.6 RF06 - Consultar número de ativações por elemento State

É necessário que o J-SDL permita consultar o contador de ativações ocorridas por instância de elemento State desde o início da simulação. Esta informação pode ser útil para o desenvolvimento de aplicativos que utilizem critérios de cobertura baseados em elementos *State.*

3.1.7 RF07 - Estimular sinais

É necessário que o J-SDL permita o estímulo de sinais como se fossem estimulados pelo ambiente onde o sistema está inserido, e que como conseqüência do estímulo de sinais toda a especificação reaja conforme as regras de comportamentos definidas no requisito RF01.

3.1.8 RF08 - Reiniciar a simulação

É necessário que o J-SDL permita reiniciar a simulação. O ato de reiniciar a simulação consiste em que todas as instâncias de elementos *Process* passem a possuir o atributo Elemento *State* atual referenciando a instância de elemento *State* referenciada pelo atributo Elemento *State* padrão. Além disso, é

necessário que os contadores de ativações relativos a cada instância de elementos *Input* e *Save* sejam reiniciados.

3.1.9 RNF01 - Independência de plataforma

Devido ao fato de se tratar de um *framework*, destinado a oferecer facilidades para a construção de outros programas, é importante que a utilização do J-SDL seja possível em várias plataformas diferentes.

3.1.10 RNF02 - Independência de software comercial.

Como o J-SDL será utilizado, principalmente, em uma instituição pública é interessante que o mesmo não possua dependência com nenhum tipo de *software* comercial, sendo possível utilizar o mesmo em um ambiente baseado apenas em soluções de *software* livre.

3.1.11 RNF013 – Possibilitar expansão e adaptação .

A arquitetura do J-SDL deve prever a necessidade de responder as modificações resultantes das próximas versões da SDL, por isso, é necessário que a arquitetura utilizada ofereça facilidades para incorporação de novos elementos e regras de comportamento.

3.2 Visão de Projeto

Devido ao formato de framework, o J-SDL foi projetado de forma a ser facilmente incorporado a aplicativos que possuam a necessidade de simular comportamento, em termos de mudança de estados, a partir de especificações de sistemas em SDL. A Figura 3.3 apresenta uma visão genérica e de alto

nível, que ilustra a utilização do J-SDL por meio de um aplicativo que possua a finalidade de geração automática de casos de teste.

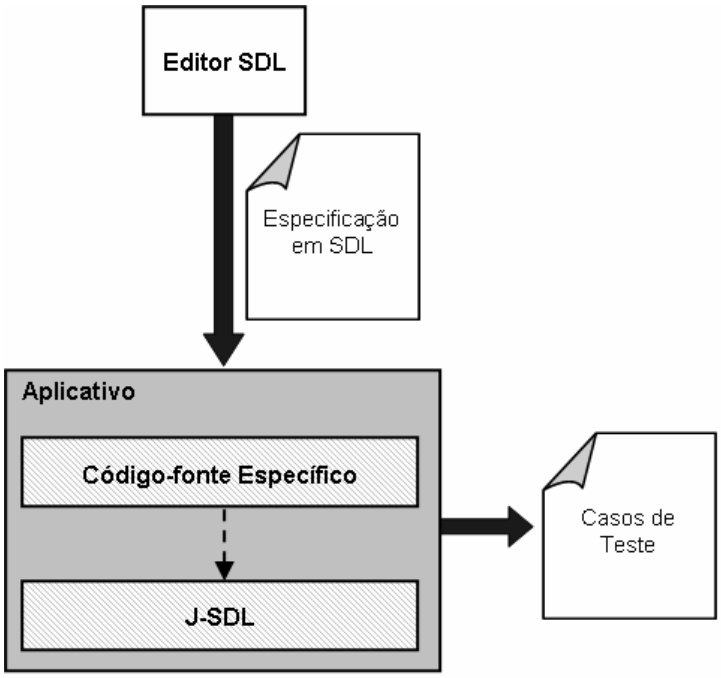

Figura 3.3 – Ambiente típico de utilização do J-SDL.

A necessidade de criar especificações em SDL é suprida pela utilização de algum editor que permita a exportação de especificações para arquivos em algum formato que favoreça o intercambio de dados, por exemplo: XML (*Extensible Markup Language*). São exemplos de ferramentas que permitem a edição de especificações em SDL: *Cinderella,* JADE, *Telelogic TAU* e *Object Geode*.

O aplicativo, em questão, carrega a especificação a partir de um arquivo que contenha a especificação em um formato de intercambio conhecido e utiliza o J-SDL para manter a especificação em memória e gerar os casos de teste, utilizando algum critério de cobertura.

A partir dos requisitos funcionais levantados pelas atividades de análise, foram projetados dois módulos internos ao J-SDL, denominados: *Model* (Modelo) e *Service* (Serviço). O módulo *Model* contém classes que mapeiam os elementos da linguagem SDL, permitindo a manipulação da especificação em memória e a simulação de comportamento. O módulo *Service* contém classes que oferecem informações de alto nível a respeito da simulação, que permitem a redução da complexidade na codificação de aplicativos que utilizem do J-SDL.

A Figura 3.4 apresenta um diagrama de pacotes da UML contendo relativos aos dois módulos do J-SDL.

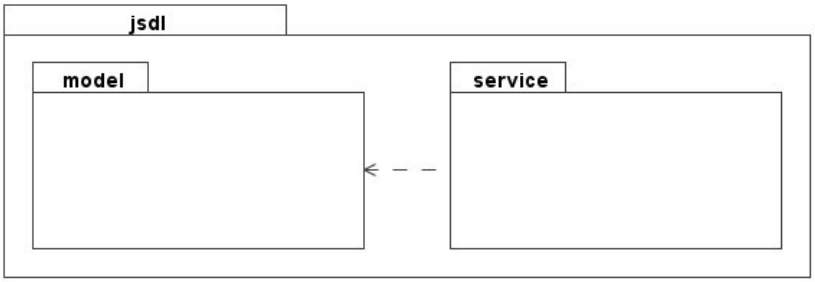

Figura 3.4 – Diagrama de pacotes referente aos módulos do J-SDL.

3.2.1 O Pacote *Model*

O pacote *Model* foi criado com o objetivo de agrupar as classes do J-SDL que representam os elementos da linguagem SDL. São essas classes que permitem a manipulação da especificação em memória e a simulação de comportamento.

O pacote *Model* foi organizado em três sub-pacotes que referenciam os tipos de elementos da linguagem SDL: *Structure* (estrutura), C*ommunication* (comunicação) e *Behavior* (comportamento). A Figura 3.5 apresenta o detalhamento do pacote *Model*.

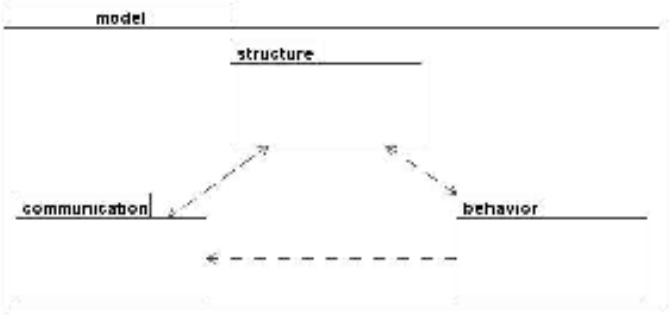

Figura 3.5 – Diagrama de pacotes referente ao pacote *Model*.

O pacote *Structure* agrupa as classes que mapeiam os elementos de estrutura: *System*, *Block* e *Process*. Além de encapsular as regras para envio e recebimento de sinais, previstas pelo levantamento de requisitos, as classes desse pacote oferecem suporte à representação de todas as características de hierarquia e paralelismo da linguagem SDL. A Figura 3.6 apresenta um diagrama de classes da UML contendo as classes desse pacote.

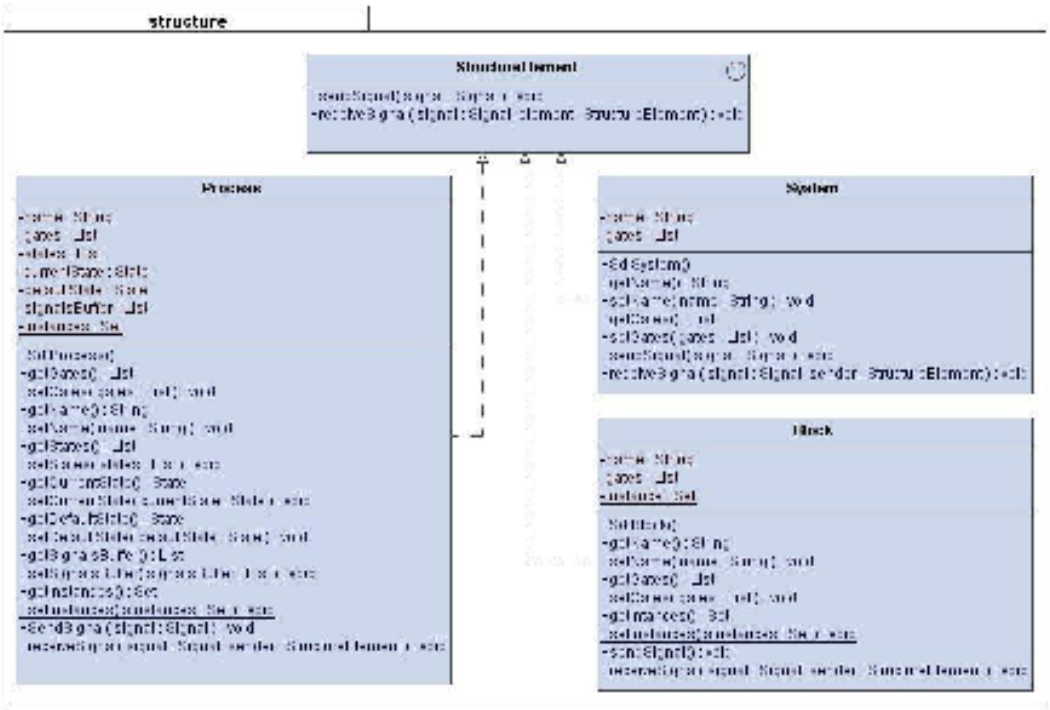

Figura 3.6 – Diagrama de classes referente ao pacote S*tructure*

Duas instâncias de elementos de estrutura (*System*, *Block* ou *Process*) podem ser conectadas com a finalidade de propagação de sinais. Essa conexão é possível através da utilização dos elementos de comunicação *Gate* e *Process*, sendo necessário que cada instância dos elementos de estrutura possua uma instância de elemento *Gate*, permitindo a interconexão por meio de uma instância de elemento *Process*.

Para facilitar a conexão entre elementos de estrutura, foi definida uma interface denominada *StructureElement.* No paradigma da programação orientada a objetos, uma interface representa uma classe abstrata (sem implementação) que define as operações que um conjunto de classes devem implementar. Dessa forma, na interface *StructureElement* foram declarados os métodos *sendSignal* e *receiveSignal*, que foram posteriormente definidos em cada uma das classes relativas a elemento de estrutura, de acordo com o comportamento apresentado pelos diagramas de seqüência da Figura 3.1 e da Figura 3.2.

Considerando que todas as classes que mapeiam elementos de estrutura são implementações de *StructureElement*, torna-se mais simples realizar a simulação de comportamento, pois em tempo de simulação, cada instância de elemento de estrutura pode enviar sinais sem se preocupar com o tipo do elemento de estrutura referente a instância que esta recebendo o sinal. Esse comportamento polimorfico não só diminui a complexidade da lógica envolvida na simulação de comportamento, como atende ao requisito RNF013 tornando o pacote Model extremamente adaptável e flexível.

No paradigma da programação orientada a objetos, poliformismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma assinatura, mas comportamentos distintos especializados para cada classe derivada, a partir de uma referência a uma instância da superclasse.

Se houver necessidade de criar uma classe para mapear um novo elemento de estrutura da SDL, basta que essa classe implemente *StructureElement* para que seja compatível com as classes pré-existentes. Se houver a necessidade de redefinir o comportamento de alguma das classes pré-existentes, basta criar uma nova classe que seja herdeira da classe original e redefinir os métodos *receiveSignal* e *sendSignal*.

Outro aspecto importante, adotado em algumas classes do pacote *Model,* é a presença de um atributo estático, pertencente à classe e não à instância da classe, denominado *Instances*. Esse atributo possui o objetivo de manter uma referência centralizada para todas as instâncias da classe a qual pertence, existentes em memória. Esse padrão de atributo funciona como facilitador para a criação das classes do pacote *Service.*

As demais classes que mapeiam os elementos da SDL foram agrupadas por meio dos pacotes: *Communication* e *Behavior*, sendo apresentadas em detalhes pela Figura 3.7 e Figura 3.8 respectivamente.

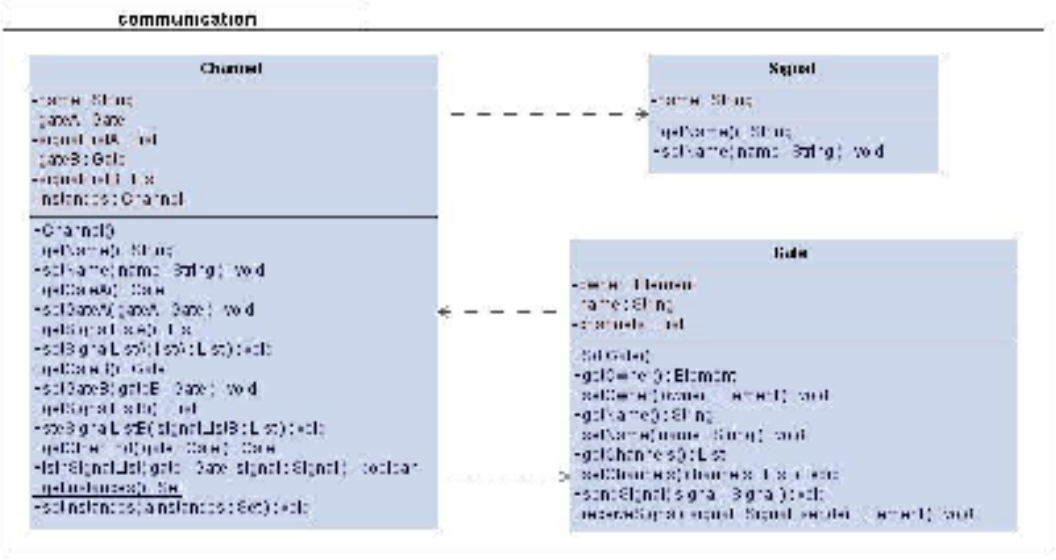

Figura 3.7 – Diagrama de classes referente ao pacote *Communication*

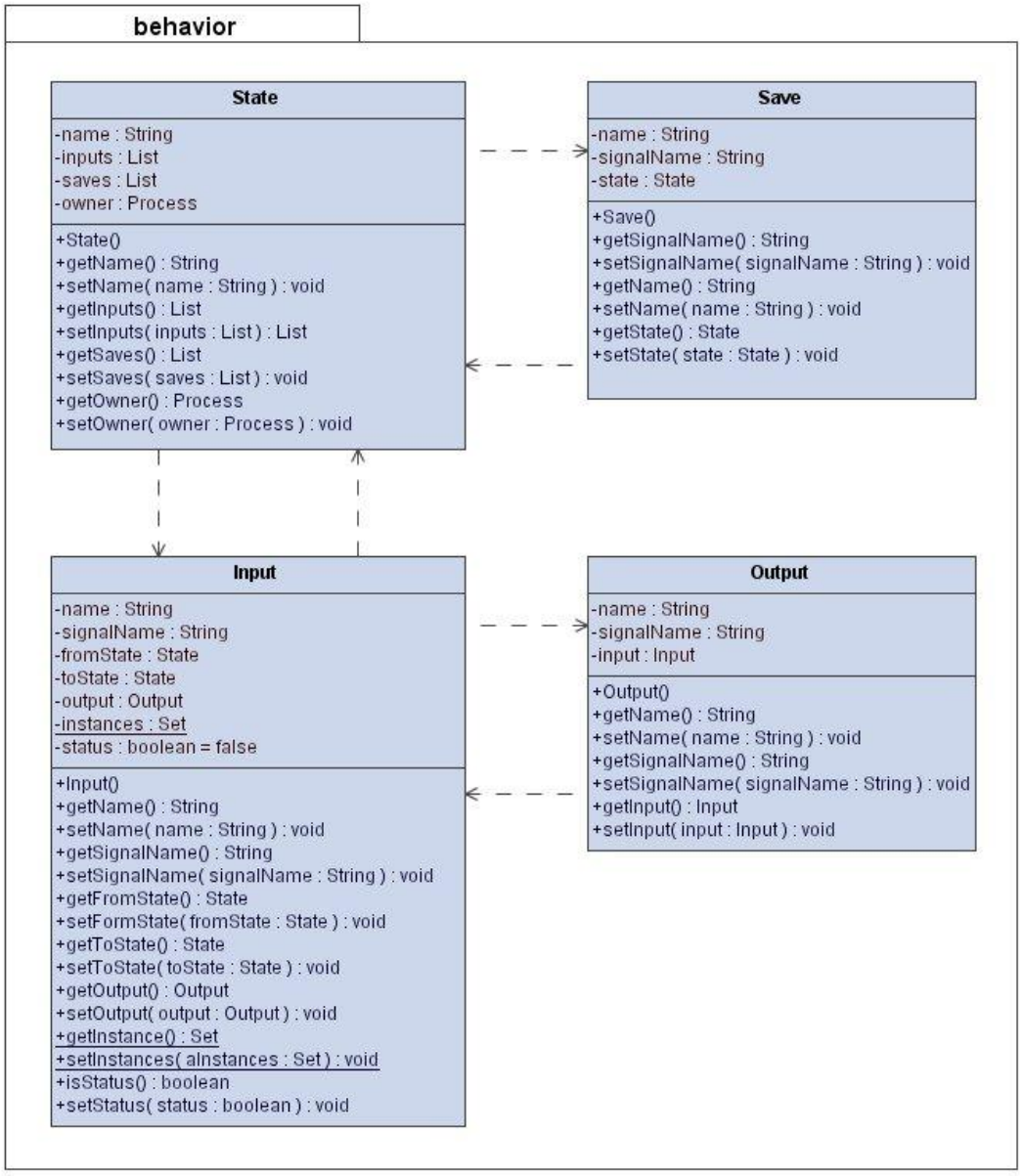

Figura 3.8 – Diagrama de classes referente ao pacote *Behavior*

3.2.2 O Pacote *Service*

O pacote *Service* agrupa as classes que oferecem informações de alto nível que permitem a simplificação do desenvolvimento dos aplicativos baseados no J-SDL. As classes desse pacote atendem aos requisitos: RF-04, RF-05, RF-06,

RF-07 e RF-08, sendo apresentadas pela Figura 3.9 por meio de um diagrama de pacotes.

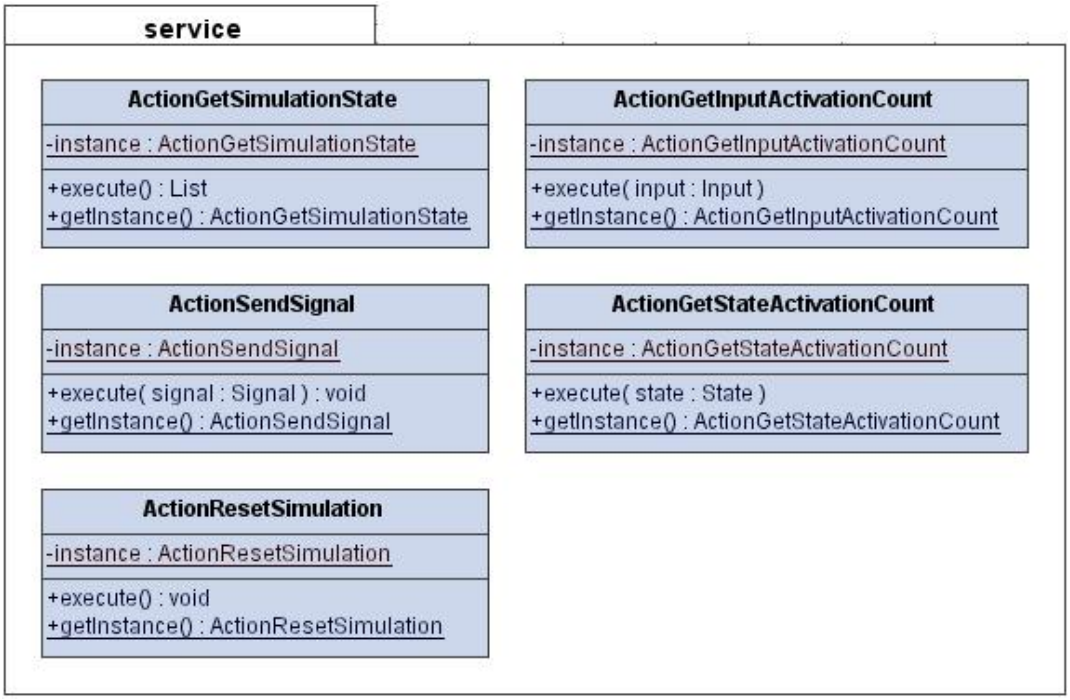

Figura 3.9 – Diagrama de classes referente ao pacote *Service.*

4 RESULTADOS

Com o intuito de validar o funcionamento do J-SDL, foram desenvolvidos dois aplicativos para geração automática de casos de teste. O primeiro aplicativo utiliza, como critério de cobertura, a ativação de todos os elementos *Inputs* da especificação que está sendo simulada, pelo menos uma vez. Este critério, baseado no método *Transition Tour*, garante uma seqüência de sinais capaz de realizar, pelo menos uma vez, todas as transições entre os estado de uma especificação, no escopo da SDL. A Figura 4.1 apresenta a lógica utilizada para codificar esse critério de cobertura a partir do J-SDL.

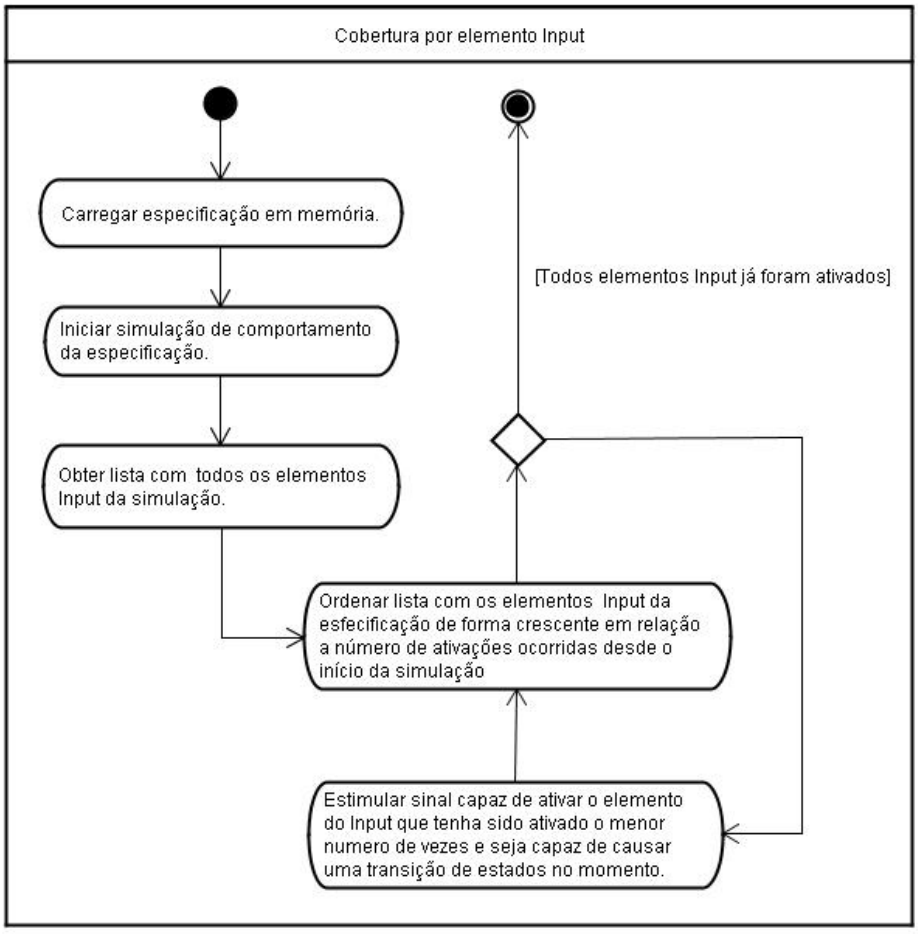

Figura 4.1 – Critério de cobertura por elemento Input.

O segundo aplicativo utiliza como critério de cobertura a ativação de todos os elementos *States* da especificação que está sendo simulada. Este critério produz uma seqüência de sinais capaz de ativar, pelo menos uma vez, todos os estados de uma especificação, no escopo da SDL. A Figura 4.2 apresenta a lógica utilizada para codificar esse critério de cobertura a partir do J-SDL.

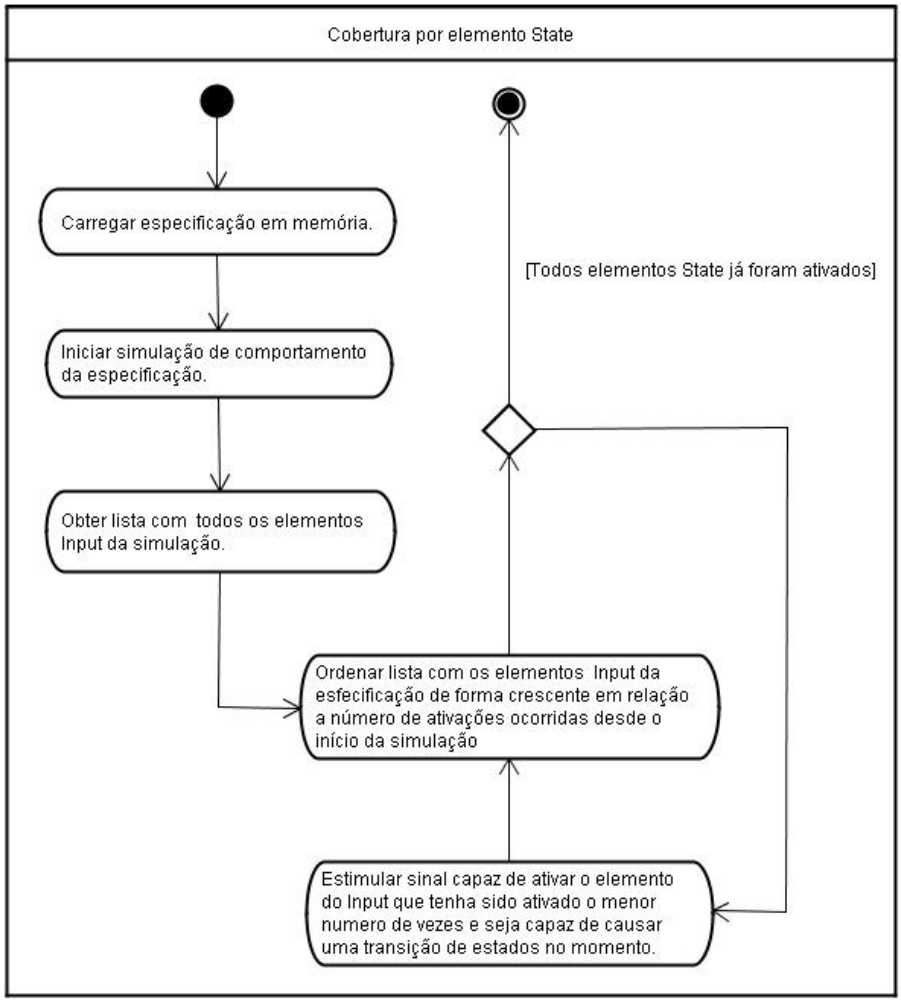

Figura 4.2 – Critério de cobertura por elemento Input.

4.1 Estudo de Caso: Sistema de Manufatura

Neste tópico será apresentado um estudo de caso realizado a partir da especificação em SDL de um sistema de manufatura, que possui internamente três componentes paralelos, sendo eles: duas máquinas que possuem a função de fabricar algum produto e um supervisor que possui a função de reparar as máquinas, caso as mesmas quebrem, podendo consertar apenas uma máquina por vez.

A Figura 4.3 ilustra o nível de sistema dessa especificação, definido por um nível de bloco denominado Principal, que se comunica com o ambiente através do canal c1 que por sua vez, conecta o sistema e o bloco Principal pelos portões a e s respectivamente. O canal c1 possui uma lista de sinais no sentido do bloco principal que permite a propagação dos sinais a1, a2, r1, r2, f1, f2, s1 e s2 para o mesmo.

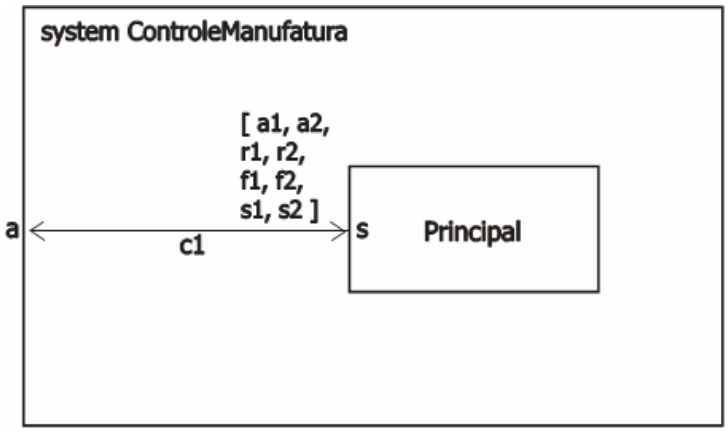

Figura 4.3 – Definição do nível de sistema do Sistema de Manufatura.

Cada um dos sinais existentes, dentro de uma especificação em SDL, mapeia um evento existente no domínio do problema. Estes eventos podem ser classificados como explícitos ou implícitos. São considerados eventos explícitos aqueles que são disparados do ambiente para o sistema, e implícitos aqueles que são disparados internamente ao sistema, normalmente em

decorrência de uma transição de estado. A Tabela 4.1 apresenta todos os sinais presentes na especificação deste sistema de manufatura, assim como, sua classificação e os eventos que os mesmos representam.

Tabela 4.1 – Sinais utilizados na especificação do Sistema de Manufatura.

Sinal	Escopo	Classificação
a1	Início da produção na Máquina1.	Explícito
a2	Início da produção na Máquina2.	Explícito
r1	Conclusão produção na Máquina1.	Explícito
r2	Conclusão produção na Máquina2.	Explícito
f1	Quebra da Máquina1.	Explícito
f2	Quebra da Máquina2.	Explícito
c1	Início do conserto da Máquina1.	Implícito
c2	Início do conserto da Máquina2.	Implícito
s1	Conclusão do conserto da Máquina1.	Explícito
s2	Conclusão do conserto da Máquina2.	Explícito

A Figura 4.4 apresenta a definição do bloco Principal, que é composto por três processos paralelos chamados: Máquina1, Máquina2 e Supervisor. Esses três processos são conectados ao bloco Principal através da porta s pelos canais c2, c4 e c3 respectivamente. Cada um desses canais possui uma lista de sinais no sentido de propagação dos processos, o que restringe a propagação dos sinais recebidos pelo bloco Principal para seus componentes. Por exemplo, quando um sinal a1 for recebido pelo sistema através da porta a ele será propagado para a porta s do bloco Principal, através do canal c1 que possui o sinal s em sua lista de sinais no sentido do bloco Principal. Dentro do bloco Principal, o sinal a1 será propagado apenas para o processo Máquina1, pois este é o único que possui um canal com uma lista de sinais que contenha o sinal a1 no seu sentido de propagação.

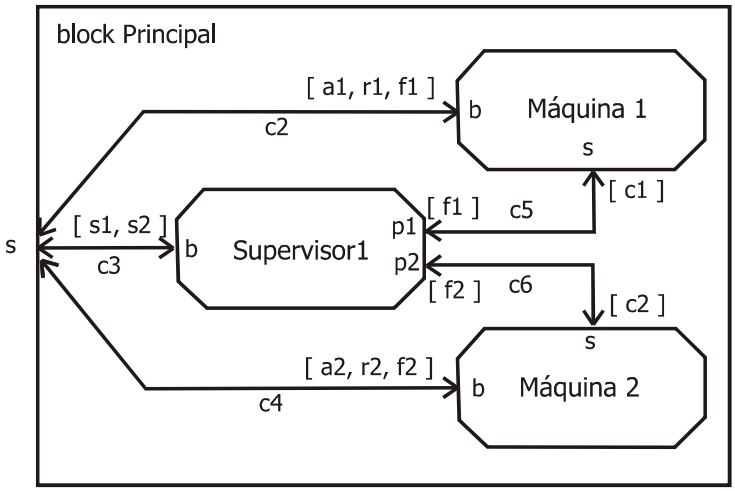

Figura 4.4 – Definição do bloco Principal do Sistema de Manufatura.

Os processos Máquina1 e Máquina2 trocam sinais com o processo Supervisor através dos canais c5 e c6 respectivamente. Pela propagação dos sinais f1 e f2, os processos Maquina1 e Máquina2 informam ao processo Supervisor quando estão quebrados. Pela propagação dos sinais c1 e c2, o processo Supervisor informa aos processos Maquina1 e Máquina2 que os mesmos foram consertados.

A Figura 4.5 e a Figura 4.6 apresentam as definições de comportamento dos processos Máquina1 e Máquina2, os quais possuem comportamentos idênticos.

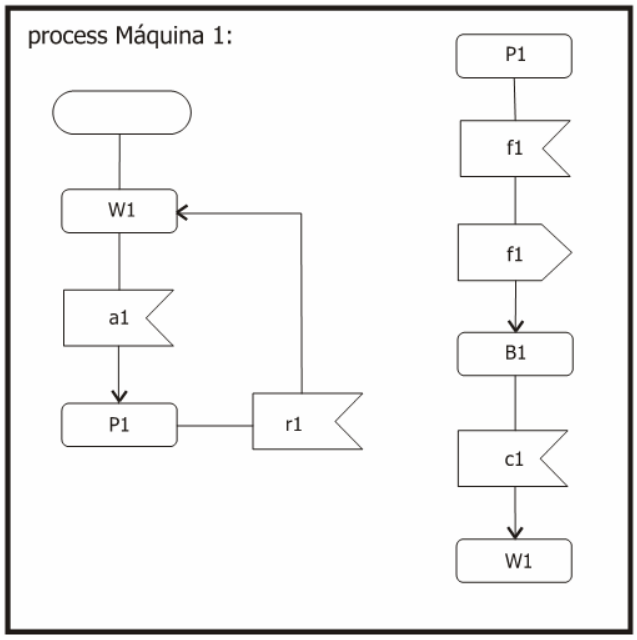

Figura 4.5 – Definição do processo Máquina1 do Sistema de Manufatura.

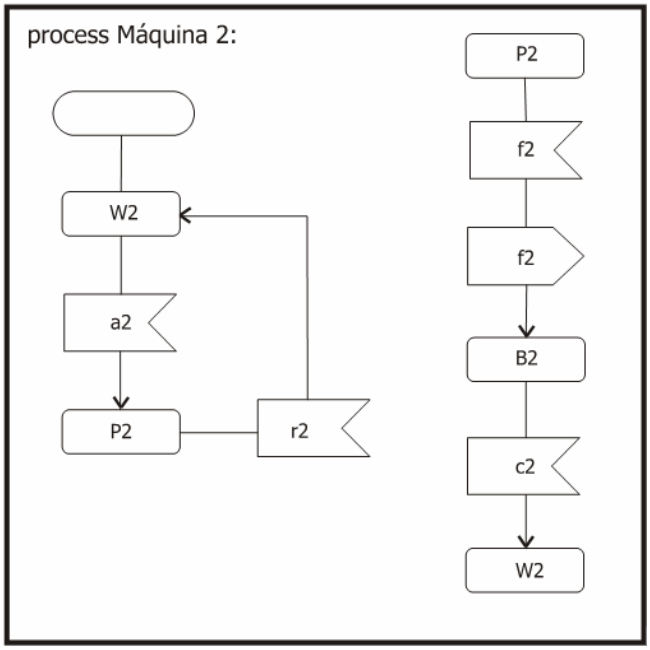

Figura 4.6 – Definição do processo Máquina2 do Sistema de Manufatura.

O processo Máquina1 possui os estados: W1, P1 e B1, sendo W1 seu estado inicial. O estado W1 possui uma entrada para o sinal a1. Uma entrada em SDL

define a possibilidade de transição de estados mediante ao recebimento de um sinal, por exemplo, quando o processo Máquina1 está no estado W1 e recebe um sinal a1, ocorre uma transição do estado W1 para o estado P1. O estado P1 possui entradas para os sinais: r1 e f1 e o estado B1 para o sinal c1. É interessante destacar que a transição do estado P1 para o estado B1 causa o envio do sinal implícito f1 que irá informar ao processo Supervisor que a Máquina1 está quebrada. A Tabela 4.2 apresenta a descrição de todos os estados possíveis para cada um dos processos dessa especificação.

Tabela 4.2 – Estados do Sistema de Manufatura.

Estado	Processo	Significado
W1	Máquina 1	Pronto.
P1	Máquina 1	Produzindo.
B1	Máquina 1	Quebrado.
W2	Máquina 2	Pronto.
P2	Máquina 2	Produzindo.
B2	Máquina 2	Quebrado.
WS	Supervisor	Pronto.
C1	Supervisor	Consertando Máquina1.
C2	Supervisor	Consertando Máquina2.

A Figura 4.7 apresenta a definição do processo Supervisor, que possui os estados WS, C1 e C2, sendo WS seu estado inicial. O estado WS possui entradas para os sinais f1 e f2. O estado C1 possui entrada para o sinal s1 e registro para o sinal f2. O estado C2 possui entrada para o sinal s2 e registro para o sinal f1. Em SDL um registro tem a função de armazenar um sinal recebido até a próxima transição de estados. Por exemplo, quando o processo Supervisor está no estado C1 e recebe um sinal f2, o mesmo é armazenado, caso o processo receba em seguida o sinal s1, uma transição para o estado WS será disparada. Quando a transição para o estado WS for concluída o processo Supervisor irá receber naquele momento, o sinal f2 que foi

armazenado anteriormente, causando, nesta situação, a transição do estado WS para C2.

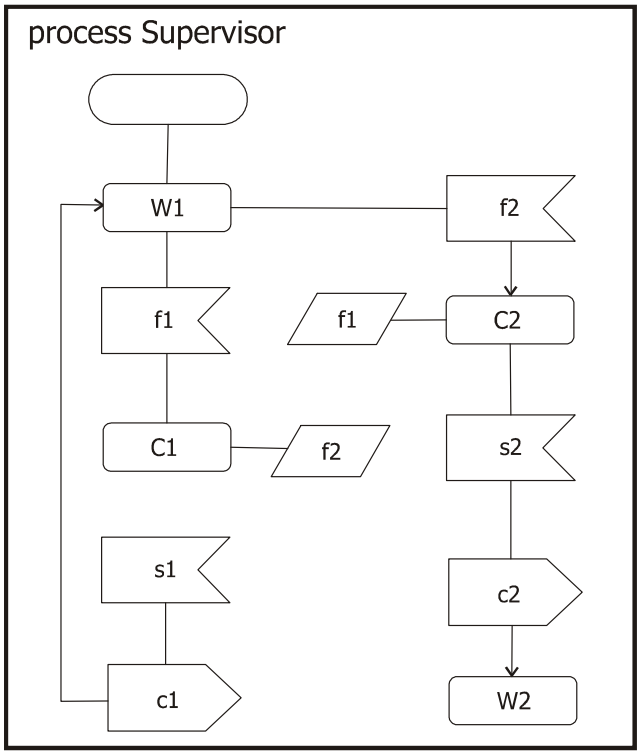

Figura 4.7 – Definição do processo Supervisor do Sistema de Manufatura..

Os casos de teste encontrados pelos dois critérios de cobertura foram idênticos para essa especificação. A convergência nos resultados encontrados ocorreu porque os dois critérios utilizam estratégias muito parecidas para selecionar o próximo sinal a ser estimulado, diferindo apenas no critério de parada, um baseado em instâncias de elementos *Input* e o outro baseado em instâncias de elementos *State.* Em especificações maiores é provável que isso não ocorra, devido ao aumento da quantidade de estados e das possibilidades de estímulo de sinais. A Tabela 4.3 apresenta os casos de testes gerados pelas duas aplicações.

Tabela 4.3 – Casos de teste para o Sistema de Manufatura.

Passo	Sinal	Estado
		W1, W2, WS
1	a1	P1, W2, WS
2	r1	W1,W2 , WS
3	a2	W1, P2, WS
4	r2	W1,W2,WS
5	a1	P1, W2, WS
6	f1	B1, W2, C1
7	s1	W1,W2, WS
8	a2	W1, P2, WS
9	f2	W1, B2, C2
10	s2	W1, W2,WS

4.2 Estudo de Caso: Sistema Produtor/Consumidor

Neste tópico será apresentado um estudo de caso realizado a partir da especificação em SDL de um sistema caracterizado por uma relação de produção e consumo. Este sistema possui internamente três componentes paralelos, sendo eles: Produtor, Consumidor e *Buffer*. O funcionamento do sistema é simples: o Produtor coloca em *Buffer* o que produz; o *Buffer* possui capacidade de armazenar apenas um produto por vez; o Consumidor só pode consumir um produto por vez a partir *Buffer*; o Produtor só pode produzir quando o *Buffer* estiver vazio; o Produtor pode quebrar.

A Figura 4.7 ilustra o nível de sistema dessa especificação, definido por um nível de bloco denominado Principal, que se comunica com o ambiente através do canal c1 que por sua vez, conecta o sistema e o bloco Principal pelos portões a e b respectivamente. O canal c1 possui uma lista de sinais no sentido do bloco principal que permite a propagação dos sinais a1, f1, c1, p1, f1, a2 e f2 para o mesmo.

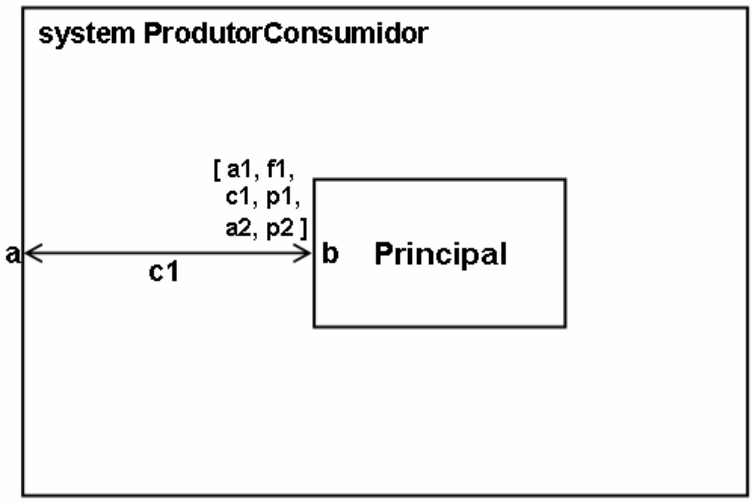

Figura 4.7 – Definição do bloco Principal do Sistema Produtor/Consumidor.

A Tabela 4.4 apresenta todos os sinais presentes na especificação deste sistema, assim como sua classificação e os eventos que os mesmos representam.

Tabela 4.4 – Sinais utilizados na especificação do sistema Produtor/Consumidor.

Sinal	**Escopo**	**Classificação**
a1	Início da produção pelo Produtor.	Explícito
f1	Quebra do Produtor.	Explícito
c1	Conserto do Produtor.	Explícito
p1	Conclusão da produção pelo Produtor.	Explícito
a2	Início do consumo pelo Consumidor.	Explícito
p2	Conclusão consumo pelo Consumidor.	Explícito
Inc	Produto colocado em *Buffer*.	Implícito
Dec	Produto removido do *Buffer*.	Implícito

A Figura 4.8 apresenta a definição do bloco Principal, através dos níveis de processo: Produtor, Consumidor e *Buffer*.

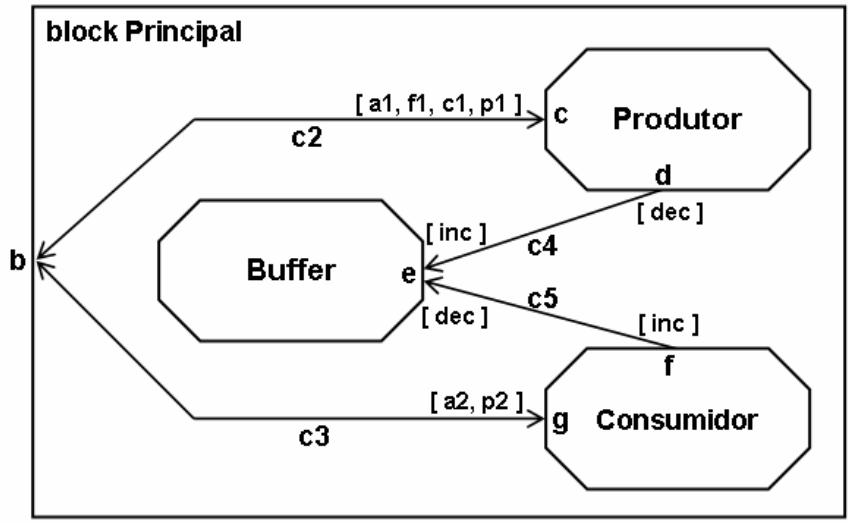

Figura 4.8 – Definição do bloco Principal do Sistema Produtor/Consumidor.

Os processos Produtor e Consumidor recém sinais através dois canais c2 e c3 que estão conectados ao portão b. Esses processos, também, trocam sinais com o processo Buffer através dos canais c4 e c5 respectivamente. O processo Produtor informa ao processo Buffer que acabou a produção de um produto por meio da propagação do sinal inc, que é propagado em seqüência pelo processo Buffer para o processo Consumidor. O processo Consumidor informa o processo Buffer que realizou o consumo de um produto por meio da propagação do sinal dec, que é propagado em seqüência pelo processo Buffer para o processo Produtor.

O processo Produtor possui os estados: O1, P1, Q1 e W1, sendo O1 seu estado inicial. A Figura 4.9 apresenta todos os detalhes de definição de comportamento do processo Produtor e a Tabela 4.4 apresenta a descrição de todos os estados possíveis para cada um dos processos dessa especificação.

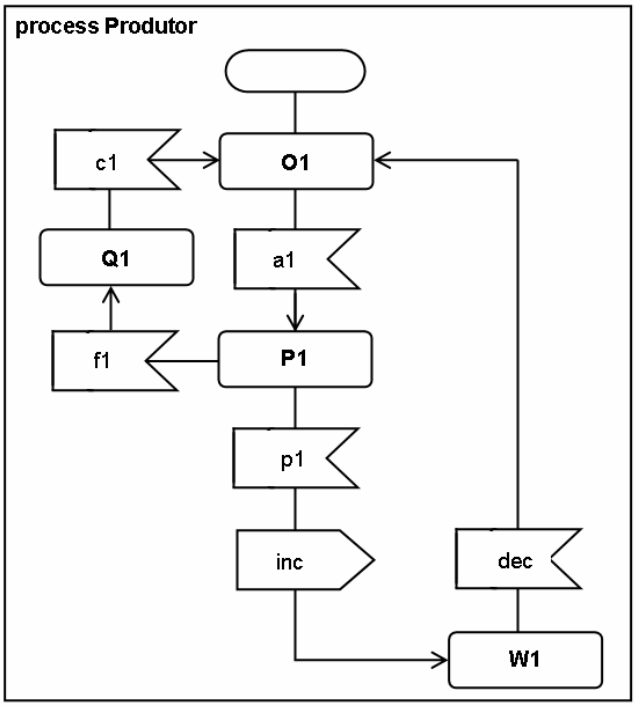

Figura 4.9 – Definição do processo Produtor do Sistema Produtor/Consumidor.

Tabela 4.5 – Estados do Sistema de Manufatura.

Estado	Processo	Significado
O1	Produtor	Ocioso.
P1	Produtor	Produzindo.
W1	Produtor	Aguardando.
Q1	Produtor	Quebrado.
W2	Consumidor	Aguardando.
O2	Consumidor	Ocioso.
P2	Consumidor	Consumindo.
B0	*Buffer*	Vazio.
B1	*Buffer*	Cheio.

O processo Consumidor possui os estados: W2 O2 e P2, sendo W2 seu estado inicial. A Figura 4.10 apresenta todos os detalhes de definição de comportamento desse processo.

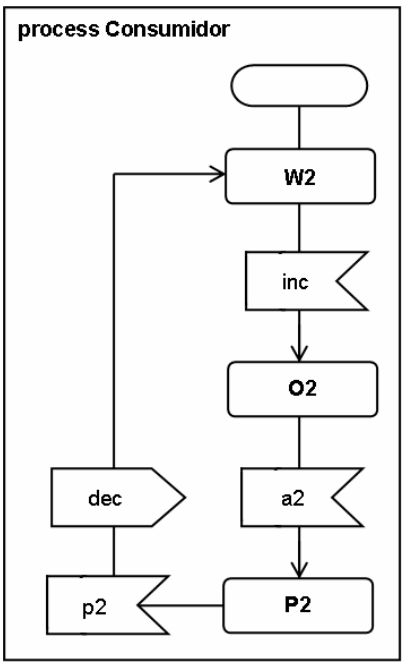

Figura 4.10 – Definição do processo Produtor do Sistema Produtor/Consumidor.

O processo *Buffer* possui os estados: B0 e B1, sendo B0 seu estado inicial. A Figura 4.11 apresenta todos os detalhes de definição de comportamento desse processo.

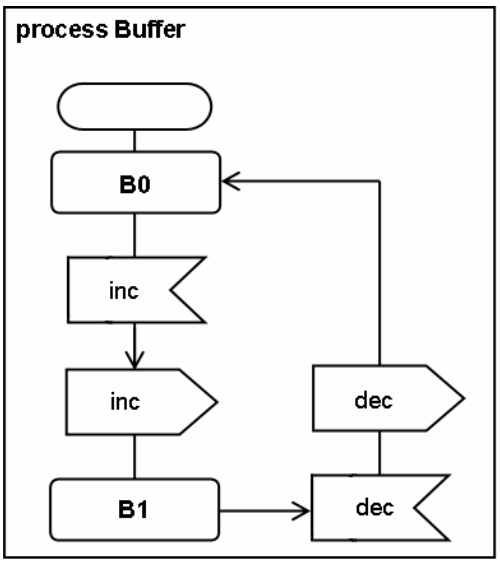

Figura 4.11 – Definição do processo *Buffer* do Sistema Produtor/Consumidor.

Novamente, os casos de teste encontrados pelos dois critérios de cobertura foram idênticos para essa especificação. A Tabela 4.6 apresenta os casos de testes gerados pelas duas aplicações.

Tabela 4.6 – Casos de teste para o Sistema Produtor/Consumidor.

Passo	**Sinal**	**Estado**
		O1, W2, B0
1	a1	P1, W2, B0
2	F1	Q1, W2, B0
3	c1	O1, W2, B0
4	a1	P1, W2, B0
5	p1	W1, O2, B1
6	a2	P2, O2, B1
7	p2	O1, W2, B0

5 CONCLUSÃO E TRABALHOS FUTUROS

A realização exaustiva de testes de software em sistemas complexos, como os de aplicação espacial, é impraticável, devido ao tempo que seria consumido pelas atividades de teste, em decorrência da complexidade desse tipo de software (SANTIAGO *et al*., 2006). Dessa forma, torna-se importante a utilização de técnicas que permitam a geração automática de casos de teste a partir de especificações formais que possam ser manipuladas por computador.

Sistemas de *software* de aplicação espacial são considerados sistemas reativos, pois podem ser representados como um conjunto de estados e transições entre estados, que ocorrem por meio de estímulos também conhecidos como eventos. MEFs são uma técnica formal de descrição, utilizada para a representação desse tipo de sistema, devido ao fato dessa técnica lidar naturalmente com estados e transições.

Faltam, na técnica de MEFs, facilidades para a representação, pelo menos de forma direta, de características comuns dos sistemas complexos modernos, tais como hierarquia e paralelismo. Por esse motivo, torna-se interessante o uso de técnicas formais de mais alto nível que ofereçam essas facilidades e possibilitem a manipulação de especificações de sistemas reativos pelo computador, como por exemplo: *Statecharts* (HAREL, 1987), (AMARAL, 2005), (SANTIAGO *et al.*, 2006), Redes de Petri (DESEL *et al.*, 2007), SDL (WONG, 2003), entre outras.

Em trabalhos anteriores realizados pelo grupo de testes do LAC e CEA do INPE, ficou demonstrada a viabilidade de gerar casos de testes de forma automática a partir de especificações em Statecharts (AMARAL, 2005), (SANTIAGO *et al*., 2006), (ARANTES *et al*., 2008), (SOUZA *et al*., 2008) e (SANTIAGO *et al*., 2008). Sendo necessário nas técnicas apresentadas por

estes trabalhos a transformação da especificação em Statecharts para MEFs antes da derivação dos casos de teste.

É de interêsse desse grupo de explorar outras técnicas formais para realização de uma análise comparativa entre técnicas. Neste sentido, esta dissertação foi desenvolvida para explorar a técnica SDL, que foi escolhida como alvo de estudo devido ao fato de ser muito utilizada no setor de telecomunicações, cujos sistemas de software embarcados também podem ser classificados como reativos.

Comparando as técnicas SDL e Statecharts, é possível concluir que apesar de possuirem perspectivas diferentes, as duas apresentam facilidades para a representação de hierarquia e paralelismo. É possível observar em SDL uma facilidade maior para a representação de sistemas baseados na troca de sinais, pois essa técnica lida de forma mais natural com essa característica.

Este trabalho apresentou um *framework*, denominado J-SDL, que consiste em uma abordagem para a simulação de comportamento de sistemas reativos a partir de especificações em SDL, útil para simplificar o desenvolvimento de aplicativos que possuam a finalidade de gerar automaticamente casos de teste, utilizando critérios de cobertura.

Para o desenvolvimento do J-SDL foi definido um modelo reativo que mapeia os elementos típicos da linguagem SDL, permitindo carregar e simular em memória especificações nessa linguagem. Além disso, o J-SDL possui um módulo de serviços que permite obter em tempo de simulação informações de alto nível que facilitam a codificação de critérios de cobertura para a geração automática de casos de teste. A principal contribuição do J-SDL foi utilizar simulação de comportamento para possibilitar a geração automática de casos de teste de forma direta, sem a necessidade de gerar MEFs.

A viabilidade da utilização do J-SDL para simulação de especificações em SDL e geração automática de casos de teste para validação de especificações e verificação de sistemas de *software,* construídos a partir dessas especificações, ficou clara a partir dos resultados obtidos pela realização dos estudos de casos. Esses resultados motivam a continuidade do trabalho, por meio do desenvolvimento de novas funcionalidades para o módulo de serviço e da integração do J-SDL a uma ferramenta *web* colaborativa para a geração automática de casos de teste (Arantes et at. 2008).

REFERÊNCIAS BIBLIOGRÁFICAS

ARANTES, A. O., VIJAYKUMAR, N. L. , SANTIAGO, V. A. , CARVALHO, A. R. **Automatic Test Case Generation through a Collaborative Web Application.** In IASTED-EuroIMSA, 2008, Innsbruck. Proceedings of the IASTED International Conference Internet & Multimedia Systems & Applications. Calgary: IASTED, p. 27-32, 2008.

AMARAL, A. S. M. S. **Geração de casos de teste para sistemas especificados em statecharts**. 2005. 162 p. (INPE-14215-TDI/1116). Dissertação (Mestrado em Computação Aplicada) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos. 2005. Disponível em: <http://urlib.net/sid.inpe.br/MTC-m13@80/2006/02.14.19.24>. Acesso em: 06 mar. 2008.

Boehm, B. W. **Software Engineering: R & D Trends and Defense Needs.** Proceedings of the Conference on Research Directions in Software Technology, Providence, Oct. 1977.

BINDER, R. V. **Testing object-oriented systems:** models, patterns and tools. Boston, MA: Addison-Wealey, 2000.

BISWAS, R.; ORT , E. **The Java Persistence API** - A Simpler Programming Model for Entity Persistence. 2006. Disponível em: <http://java.sun.com/developer/technicalArticles/J2EE/jpa/ >. Acesso em: 30 mar. 2008.

DESEL, J., OBERWEIS, A., ZIMMER, T. **A test case generator for the validation of high-level Petri nets.** In INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION PROCEEDINGS, 6., p. 327 – 332, 1997.

ELLSBERGER, I.; HOGREFE, D.; SARMA, A. **SDL: Formal object-oriented language for communicating systems.** Prentice Hall Europe, 1997. ISBN(0-13-632886-5).

HAREL, D. **Statecharts:** a visual formalism for complex systems**.** Science of Computer Programming, v. 8, p. 237-274, 1987.

IEEE. **Standard Glossary of Software Engineering Terminology 610.12-1990.** In IEEE Standards Software Engineering, 1999 Edition, Volume One: Customer and Terminology Standards. IEEE Press, 1999.
ITU. **ITU-T recomendation Z100**: Specification and Description Language SDL. 2002. 206 p. Disponível em: <http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf>. Acesso em: 19 mar. 2008.

LEE, D. e YANNAKAKIS, M. **Principles and Methods of Testing Finite State Machines** – A Survey. In Proceedings of the IEEE, p. 84-88, 1996.

MARTINS, E., SABIÃO, S. B. e AMBRÓSIO, A. M. **ConData**: a tool for automating specification-based test case generation for communication systems. In HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 33., Maui, USA. **Proceedings…** [S.l: s.n.], 2000.

MYERS, G. J. **The art of software testing.** John Wiley &Sons, 1th edition, New York, USA, 1979.

PIMONT, S. e RAULT, J.C. **An approach towards reliable Software.** In INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 4., Munich, Germany, p. 220-230, 1979.

PRESSMAN, R. S. **Engenharia de software.** 6.ed. São Paulo: McGraw-Hill, 2006.

SANTIAGO, V.; AMARAL, A. S. M. S.; VIJAYKUMAR, N. L.; MATTIELLOFRANCISCO,M. F.; MARTINS, E.; LOPES, O. C. **A pratical approach for automated test case generation using statecharts.** In: INTERNATIONAL WORKSHOP ON TESTING AND QUALITY ASSURANCE FOR COMPONENTBASED SYSTEMS, 2. (TQACBS 2006), 2006, Chicago. **Proceedings...** 2006.(INPE-14044--PRE/9218). Disponível em: <http://urlib.net/sid.inpe.br/mtcm16@80/2006/08.10.12.08>. Acesso em: 10 abr. ,2008.

SANTIAGO, V.; VIJAYKUMAR, N. L.; GUIMARÃES, D.; AMARAL, A. S.; FERREIRA, E**. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models.** In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING VERIFICATION ND VALIDATION (ICST 2008), 1., 2008, Lillehammer, Norway. **Proceedings...** [S.l: s.n.], 2008.

SOMMERVILLE, I. **Software engineering.** 6ª Edição. Harlow: Addison Weley, 2003. 592 p.

SOUZA, E.F.; SANTIAGO, V.; GUIMARÃES, D.S.; Vijaykumar, N.L. **Evaluation of Efficiency of Test Criteria for Space Application Software Modeling.** In STATECHARTS. INTERNATIONAL CONFERENCE ON INNOVATION IN SOFTWARE ENGINEERING (ISE), Vienna, Austria, 2008.

WONG, W. E., SUGETA T., LI J. J. e MALDONADO J. C. **Coverage testing software architectural design in SDL.** Journal of Computer Networks, Vol. 42, Issue 3, p. 359-374, 1996.

BIBLIOGRAFIA RECOMENDADA

HENNIGER, O., HASAN, U. **Test generation based on control and data dependencies within multi-process SDL specifications**. In WORKSHOP OF THE SDL FORUM SOCIETY ON SDL AND MSC, 2., (SAM 2000), Grenoble, França, 2000.

SCHIMITT, M., EK, A., GRABOWSKI J., HOGRETE, D. e KOCK, B. **Autolink** – Putting SDL-based test generation into practice. In IFIP TC6 INTERNATIONAL WORKSHOP ON TESTING COMMUNICATING SYSTEMS, 11., p.227-244, 1998.

WONG, W. E., SUGETA T., QI Y. e MALDONADO J. C. **Smart debugging software architectural design in SDL.** Journal of Systems and Software, Volume 76, Number 1, pp. 15-28, 2005.

WONG, W. E., SUGETA T. e MALDONADO J. C. **Structural and Mutation Testing for SDL Specifications:** A Case Study. In Proceedings of The 6th IEEE Latin-American Test Workshop (LATW), Salvador, Bahia, Brasil, 2005.

WONG, W. E., SUGETA T. e MALDONADO J. C. **Mutation Testing Applied to Validate SDL Specifications.** In Proceedings of The 16th IFIP International Conference on Testing of Communicating Systems (TestCom), Oxford, United Kingdom, p.193-208, 2004.