ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DEPARTAMENTO DE ENGENHARIA MECATRÔNICA E DE SISTEMAS MECÂNICOS

FERRAMENTA BASEADA EM REDE DE PETRI PARA MODELAGEM, SIMULAÇÃO, PROGRAMAÇÃO E SUPERVISÃO DE SISTEMAS DE AUTOMAÇÃO

Renato Gonçalves de Freitas
Victor Anselmo Silva

São Paulo
2006

FICHA CATALOGRÁFICA

Freitas, Renato Gonçalves de
Ferramenta gráfica baseada em rede de Petri para modelagem, simulação, programação e supervisão de sistemas de automação / R.G. de Freitas, V.A. Silva. -- São Paulo, 2006.
84 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Controladores programáveis 2.Redes de Petri 3.Softwares (Modelagem; Simulação) I.Silva, Victor Anselmo II.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos III.t.

Freitas, Renato Gonçalves de
Ferramenta gráfica baseada em rede de Petri para modelagem, simulação, programação e supervisão de sistemas de automação / R.G. de Freitas, V.A. Silva. -- São Paulo, 2006.
84 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Controladores programáveis 2.Redes de Petri 3.Softwares (Modelagem; Simulação) I.Silva, Victor Anselmo II.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos III.t.

Aos meus pais e minha irmã, que sempre me incentivaram nos meus trabalhos.

Renato Gonçalves de Freitas

Ao meu avô, Samuel Anselmo, e à minha mãe, Marisa Anselmo, que desde os meus primeiros dias de vida têm me dado suporte, força e têm aguardado por esse momento. E ao meu irmão Heitor, um filho e um irmão exemplar.

Victor Anselmo Silva

AGRADECIMENTOS

Ao nosso orientador, Prof. Dr. Paulo Eigi Miyagi, pelo direcionamento e atenção dados ao longo do desenvolvimento deste trabalho de formatura.

Ao Eng. doutorando Marcosiris Amorim de Oliveira Pessoa que nos auxiliou, na fase final deste trabalho, na realização dos estudos de caso.

À Verônica, minha namorada, pela paciência, apoio e carinho demonstrados por mim nestes últimos três anos *(Victor)*.

Aos nossos professores, que contribuíram para nossa formação na Escola Politécnica da Universidade de São Paulo.

"Não existe trabalho ruim. O ruim é ter que trabalhar."

Seu Madruga (Don Ramon)

RESUMO

Os chamados sistemas a eventos discretos (SEDs), isto é, sistemas cuja dinâmica é dirigida essencialmente pela ocorrência de eventos instantâneos e estados discretos, apresentam características como concorrência, sincronismo, assincronismo e conflitos entre processos e podem ser modelados e analisados por meio de uma ferramenta gráfica e matemática chamada Rede de Petri (RdP).

Tendo como foco a implementação de sistemas de automação de SEDs com controladores programáveis (CPs), o objetivo do presente trabalho é a construção de uma ferramenta baseada em RdP para modelagem, simulação, programação e supervisão destes sistemas.

Por meio de uma interface gráfica, a ferramenta permite ao usuário criar modelos em RdP ou editar modelos existentes. Como formato de intercâmbio entre *softwares* que trabalham com RdP, esses modelos são traduzidos pela ferramenta para a notação *XML* (*eXtensible Markup Language*), a qual baseia-se em *tags* que estabelecem um procedimento para identificar, categorizar e organizar informações.

Esta ferramenta também é utilizada para traduzir o modelo em RdP de uma estratégia de controle para um programa escrito em linguagem adequada à implementação em CPs. Assim, este programa pode ser carregado e executado diretamente em um CP, responsável pela automação do sistema. Além disso, durante a operação efetiva do sistema, os sinais de entrada e de saída tratados pelo CP podem ser monitorados pela ferramenta de modo que é possível acompanhar a dinâmica do sistema, indicada pela evolução dos estados da RdP apresentada na interface.

Palavras-chave: Controladores programáveis, Redes de Petri, modelagem, simulação.

ABSTRACT

Discrete events systems (DES) are systems whose dynamic is essentially driven by the occurrence of instantaneous events and discrete states. They present characteristics as concurrency, synchronism, asynchronism and conflicts among processes, and can be modeled and analyzed through a graphical and mathematical tool called Petri Net (PN).

Considering as focus the implementation of automation systems of DESs with programmable controllers (PCs), the objective of the present work is the development of a tool based on PN for modeling, simulation, programming and supervision of these systems.

Through a graphical interface, the tool allows the user to create models in PN or edit existing models. As a format of data interchange among softwares that work with PN, the models are translated by the tool into XML (eXtensible Markup Language) notation, which is based on tags that establish a procedure to identify, categorize and organize information.

This tool is also used to translate the PN model of a control strategy for a PC program written in suitable language. Thus, this program can be directly loaded and executed in a PC, responsible for the automation of the modeled system. Throughout the effective operation of the system, the tool can monitor input and output signals treated by the PC, then, it is possible to follow/visualize the dynamics of the system, indicated by the evolution of the states of the PN presented in the interface.

Keywords: Programmable controllers, Petri Nets, modeling, simulation.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

1.1. Motivação e Justificativa

Segundo MIYAGI (1996), *man made systems* (sistemas feitos pelo homem), como sistemas de manufatura, de transporte, de comunicação, de redes de computadores, etc. podem ser caracterizados por uma dinâmica causada pela ocorrência de eventos discretos e em muitos casos são tratados como objetos de controle de sistemas de automação comandados por controladores programáveis (CPs).

Em linhas gerais, os SEDs são caracterizados por apresentarem comportamento dinâmico governado pela ocorrência de eventos discretos, vinculados a condições pré-estabelecidas, não havendo, necessariamente, uma pré-determinação do instante em que cada evento ocorre. Nesses sistemas, as transições de estados ocorrem em instantes de tempo discretos e assíncronos, em resposta a eventos considerados instantâneos (ARAKAKI, 1993).

Os SEDs incluem desde sistemas relativamente simples e de pequeno porte até sistemas de grande porte, com elementos dispostos geograficamente e com interações e funções de alta complexidade em atendimento a critérios de desempenho, produtividade e qualidade. Observa-se, no entanto, que se alguns desses critérios são mais claros em sistemas de manufatura, o mesmo não acontece em áreas como de automação residencial.

Do ponto de vista de modelagem e análise de SEDs, uma das técnicas de comprovada eficiência é a Rede de Petri (RdP). De fato, além dos vários trabalhos já publicados que exploram a aplicação de RdP na representação e aprimoramento de SEDs, a RdP foi também adotada como a base de uma linguagem de descrição funcional de CPs, o *SFC* (*Sequential Flow Chart*).

Existem assim, atualmente, diversas ferramentas de edição e análise de SEDs baseadas em RdP, bem como trabalhos que estabelecem uma relação entre grafos de RdP e programas de controle de CPs. Um exemplo disso pode ser encontrado em ZHURAWASKI & ZHOU (1994), no qual é apresentada uma

comparação entre redes de Petri de Tempo Real (*RTPN – Real Time Petri Nets*) e diagramas elétricos de relés (*LLD – Ladder Logic Diagram*).

Contudo, os arquivos gerados por estas ferramentas, em geral, estão em um formato que impossibilita o intercâmbio com outras ferramentas de modelagem e análise de SEDs e que também não é próprio para serem diretamente carregados como programas de CPs.

1.2. Objetivo

Tendo como foco a implementação de sistemas de automação de SEDs comandados por CPs, o objetivo do presente trabalho é a construção de uma ferramenta, baseada em RdP, para modelagem, simulação, programação e supervisão destes sistemas.

Por meio de uma interface gráfica, a ferramenta deve permitir ao usuário a criação de modelos em RdP ou a edição de modelos existentes. Como formato de intercâmbio entre *softwares* que trabalhem com RdP, esses modelos devem ser traduzidos pelo *software* para a notação *XML* (*eXtensible Markup Language*), a qual baseia-se em *tags* que estabelecem uma técnica para identificar, categorizar e organizar informações em arquivos de dados.

Esta ferramenta gráfica também deve ser capaz de traduzir o modelo em RdP de uma estratégia de controle para um programa para CPs, escrito em linguagem adequada à implementação. Assim, este programa poderia ser carregado diretamente em um CP, responsável pela execução do controle do sistema. Além disso, quando da execução, os sinais de entrada e de saída tratados pelo CP também devem ter a possibilidade de serem monitorados através da ferramenta de modo que se possa acompanhar a dinâmica do sistema, por meio da evolução dos estados da RdP apresentada na interface gráfica para o usuário.

2. CONCEITOS FUNDAMENTAIS

2.1. Sistema a Eventos Discretos (SED)

Um Sistema a Eventos Discretos (SED) é um sistema em que as variáveis de estado variam bruscamente em instantes determinados e tal que os valores das variáveis nos estados seguintes podem ser obtidos por meio de cálculos, diretamente a partir dos valores precedentes, não sendo necessário considerar o intervalo de tempo entre os instantes analisados (VALLETE, 1997). A figura 2.1 apresenta um gráfico ilustrativo do comportamento de uma variável de um SED ao longo do tempo.

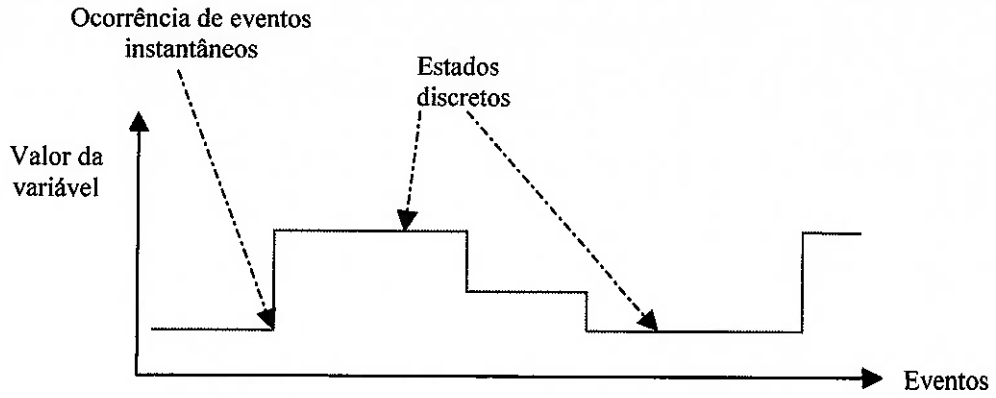


Figura 2.1: Comportamento de uma variável de um SED.

Outra definição para um SED, estabelecida por RAMADGE & WONHAM (1989), caracteriza este tipo de sistema como sendo construído e concebido pelo homem e cuja dinâmica é regida pela ocorrência de eventos discretos a intervalos em geral irregulares e desconhecidos.

Dentre algumas particularidades de SEDs, GUSTIN (1999) cita: sincronização, assincronismo, concorrência, causalidade, conflito entre processos e compartilhamento de recursos.

2.2. *Sistema de Manufatura (SM)*

Como um SED, um sistema de manufatura (SM) é decomposto por duas partes principais: o sistema físico e o sistema de controle (DICESARE et. Al, 1993). O sistema físico abrange o conjunto de recursos os quais operam sobre os materiais e sobre os processos em trabalho. Como exemplos, têm-se os tornos e fresadoras de comando numérico, unidades de produção, sistemas de transporte (transportadores, veículos automatizados, gruas, etc.), mão-de-obra humana, estoques, estações de carga e descarga, estações de controle de qualidade.

O sistema de controle é responsável pela tomada de decisão com relação aos processos e atividades que a parte física realiza. O sistema de controle otimiza a produção de um SM baseando-se em critérios de produtividade previamente estabelecidos. Como exemplo, considere um sistema de estoque de dois tipos de materiais os quais são utilizados conjuntamente para a produção de um determinado componente. Os níveis de estoque destes materiais devem ser monitorados, evitando-se assim a interrupção na produção para reposição de material e, além disso, a quantidade de cada um necessária à produção do componente deve ser medida por dispositivos auxiliares. Ao sistema de controle, neste exemplo, seriam atribuídos os comandos relativos à função de verificação periódica dos níveis de material em estoque e também dos comandos relativos à verificação da quantidade ideal de cada material necessária à produção do componente.

Para ZHOU & VENKATESH (1998), um sistema automatizado de manufatura pode ser definido como um sistema que trabalha com processamento distribuído de dados e fluxo automático de material, utilizando máquinas controladas por computador, unidades de montagem, robôs industriais, máquinas de inspeção, manipulação e movimentação de materiais e estoques todos devidamente integrados por computador. Como exemplo de um SM, a figura 2.2 esquematiza, simbolicamente, um sistema para usinagem de uma peça.

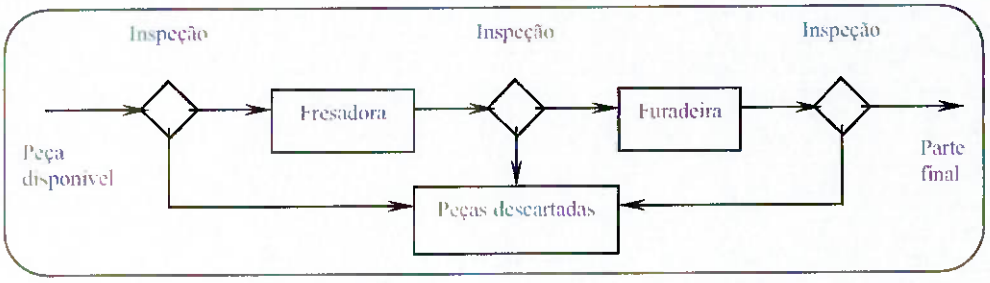


Figura 2.2: Representação para um sistema de manufatura.

Nesta representação há duas operações de usinagem envolvidas: fresamento e furação; três estações de inspeção: uma para verificação da peça que chega para ser usinada e outras duas para verificação de especificações após o fresamento e a furação. Caso alguma inspeção indique um resultado inadequado, a peça é descartada.

2.3. *Rede de Petri (RdP)*

O conceito de rede de Petri (RdP) foi introduzido por Carl Adam Petri em sua tese de doutorado (1962), como ferramenta para descrever relações entre condições e eventos. A RdP é uma ferramenta matemática e gráfica que possibilita um formalismo adequado para a modelagem, análise e projeto de SEDs (ZURAWSKI & ZHOU, 1994).

Uma RdP é um grafo bipartido composto por dois tipos de nós: lugares[1] e transições. Os lugares são representados graficamente por círculos, e as transições por barras. Lugares são conectados às transições (e transições são ligadas a lugares) por meio de arcos orientados. Arcos que partem de um lugar devem, necessariamente, chegar a uma transição e, arcos partindo de uma transição devem chegar a um lugar. Um arco que ligue dois lugares ou duas transições não é permitido. Uma marcação em uma RdP determina um número inteiro de marcas as quais são distribuídas entre os lugares. As marcas distribuídas entre os lugares identificam um estado discreto do sistema modelado. A definição formal de uma RdP é dada a seguir.

[1] Termos referentes à RdP estão em fonte Times New Roman.

Definição (VILLANI, 2004) – Uma RdP marcada é um par $\mu =< R, M_0 >$, onde:

- R é uma RdP definida pela 4-tupla $< P, T, pre, pos >$, onde:

  - $P = \{p_1, p_2, ..., p_n\}$, é um conjunto finito de lugares.

- $T = \{t_1, t_2, ..., t_n\}$, é um conjunto de transições.

- $P \cap T = \phi, P \cup T = \phi$.

- pre: $P \times T \rightarrow N$ define os arcos de entrada das transições (N é o conjunto dos números naturais).

- pos: $T \times P \rightarrow N$ define os arcos de saída das transições.

- $M_0 : P \rightarrow N$ é marcação inicial da rede.

As representações de pre e pos são feitas sob a forma de matrizes, em que as linhas correspondem aos lugares e as colunas às transições. O valor de cada componente da matriz representa o peso de um arco orientado ligando um lugar e uma transição, caso tal componente tenha valor zero, interpreta-se como inexistente o arco que liga o lugar e a transição correspondentes a este componente.

O comportamento dinâmico da RdP se dá por meio da ocorrência de eventos que movimentam as marcas entre os lugares da rede. Os eventos são as transições, e a ocorrência de um evento é interpretada como o disparo de uma transição. Uma transição está habilitada para disparo quando os lugares à sua entrada (lugares p_j, onde $pre(p_j, t_i) > 0$) possuem um número de marcas $M(p_j) \geq pre(p_j, t_i)$. Quando uma transição t_i dispara, removem-se de cada lugar p_j de entrada da transição $pre(p_j, t_i)$ marcas, e a cada lugar p_j de saída da transição são adicionadas $pos(p_j, t_i)$ marcas (VILLANI, 2004).

Segundo MURATA (1989), em sistemas modelados por RdPs os lugares e as transições podem assumir diversas interpretações (vide tabela 2.1). A presença de uma marca em um lugar, ao qual se associa uma condição, torna esta condição satisfeita. Em uma outra interpretação, um número k inteiro de marcas colocadas em um lugar indicaria k tipos de dados (informações, valores) ou k tipos de recursos (peças, ferramentas) disponíveis.

Tabela 2.1: : Algumas interpretações típicas para lugares e transições.

Lugares de entrada	Transição	Lugares de saída
Pré-condições	Evento	Pós-condições
Dados de entrada	Passo computacional	Dados de saída
Sinais de entrada	Processador de sinal	Sinais de saída
Recursos necessários	Tarefa ou trabalho	Recursos liberados

Tipos de RdPs

A literatura apresenta várias classes de RdP, tais como (ZHOU & VENKATESH, 1998; MIYAGI, 1996):

- Redes de Petri Coloridas (*CPN – Coloured Petri Nets*);
- Redes de Petri Temporizadas (*TPN – Timed Petri Nets*): redes que modelam SEDs cujas especificações de tempo para a ocorrência de um evento são importantes;
- Redes de Petri de Tempo Real (*RTPN – Real Time Petri Nets*): redes às quais se associam temporizações, tal como em uma *RTPN*, e sinais de *I/O* (entrada/saída);
- Redes de Petri Acíclicas (*APN – Acyclic Petri Nets*): redes que não apresentam *loops* na estrutura de grafo. São usadas para representar alguns tipos de processos como de manufatura, de montagem e desmontagem de peças ou componentes.

Entretanto, com base nos estudos realizados e nos objetivos considerados do presente trabalho, apresenta-se a seguir os tipos de RdP que descrevem as principais propriedades desta técnica e que foram considerados para o desenvolvimento da ferramenta de modelagem, simulação, programação e supervisão.

A) Rede Condição-Evento (CE)

Dá-se o nome de rede Condição-Evento (CE) à rede que só permite lugares com marcação binária, isto é, cada lugar da RdP pode conter apenas uma única marca, ou nenhuma (REALI, 2001). Neste tipo de rede, as transições são eventos que, ao ocorrerem, alteram o estado da rede. Os lugares são condições que, quando preenchidos por uma marca, denotam que a condição associada está satisfeita. A figura 2.3 ilustra uma rede CE.

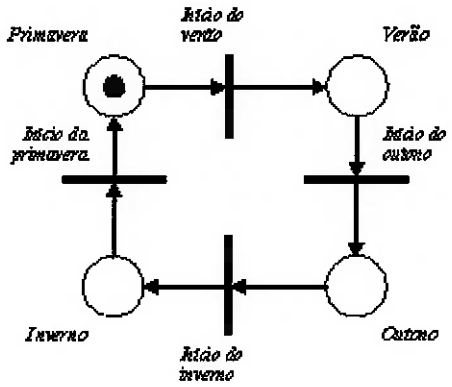


Figura 2.3: Exemplo de uma RdP CE.

Na rede da figura 2.3, a condição ***Primavera*** está satisfeita. Para que haja um disparo, um evento deve estar habilitado e, cada um dos lugares (condições) à saída deste evento não pode conter uma marca. Observando-se a figura 2.3, o evento ***Início do verão*** pode ser disparado. Após o disparo deste evento, uma marca é adicionada ao lugar (condição) ***Verão***, tornando está condição satisfeita e, conseqüentemente, habilitando a ocorrência do evento ***Início do Outono***. Os estados da rede mudam de acordo com o disparo de cada evento.

B) Rede Lugar-Transição (LT)

Em uma rede Lugar-Transição (LT), os lugares podem ter mais de uma marca e os arcos são ponderados, isto é, possuem um valor inteiro (peso) que indica quantas marcas são retiradas e/ou inseridas com o disparo da transição. O lugar em uma rede LT possui uma capacidade, isto é, um valor inteiro que denota o número máximo de marcas que o lugar pode conter. Para que haja um disparo em uma rede LT, de acordo com MIYAGI (1996) duas regras devem ser obedecidas:

(1) Para cada lugar *p* do pré-conjunto de uma transição *t*, o número de marcas em *p* não pode ser inferior ao peso do arco de *p* para *t*;

(2) Para cada lugar *p* do pós-conjunto de *t*, o peso do arco de *t* para *p* somado ao número de marcas contidas em *p* não pode exceder a capacidade de *p*.

O disparo de uma transição causa o decremento (dado pelo peso do arco) de marcas nos lugares pertencentes ao pré-conjunto da transição, e um incremento de marcas (também segundo o peso do arco) nos lugares pertencentes ao pós-conjunto da transição. A figura 2.4 apresenta uma rede LT para análise de situações em que o disparo pode ou não ocorrer.

Inicialmente, convém esclarecer que a capacidade de cada lugar é definida na figura 2.4 por meio do valor da constante K, indicada junto a cada lugar. Observando a parte (a) desta figura, nota-se que a transição está habilitada, pois as regras de disparo (1) e (2) apresentadas são obedecidas.

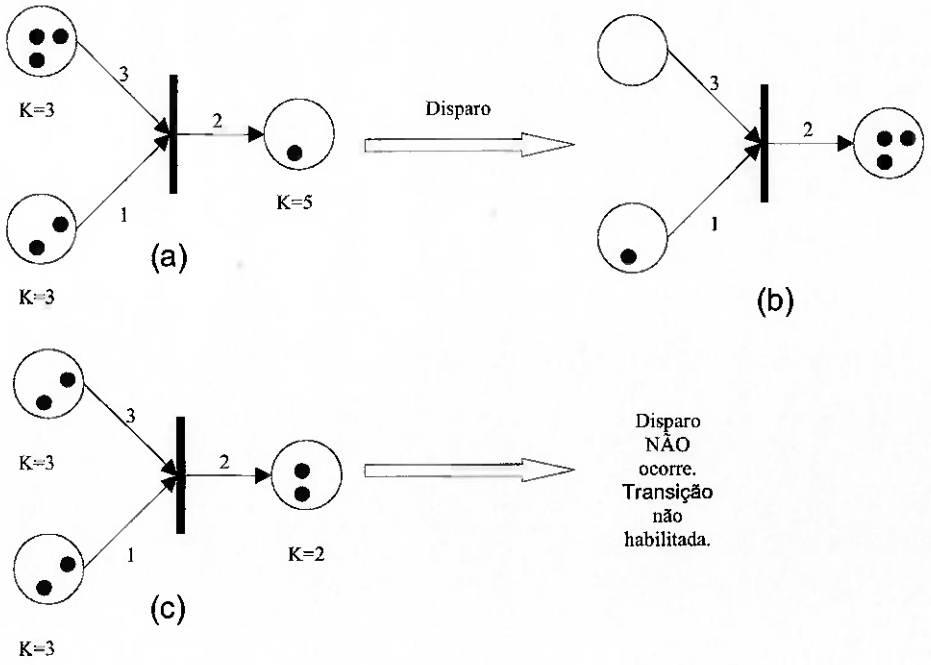


Figura 2.4: (a) Transição habilitada. (b) Estado após o disparo. (c) Situação em que o disparo não ocorre.

Desta forma, a transição dispara e o novo estado é ilustrado na parte (b) da figura. No entanto, ao aplicarem-se as regras (1) e (2) para rede da parte (c) da figura, conclui-se que estas não são atendidas e, conseqüentemente, o disparo não ocorre.

C) Rede de Sinal Interpretado

Uma RdP de Sinal Interpretado (*SIPN – Signal Interpreted Petri Net*), segundo FREY (2000), é descrita por uma 9-tupla $SIPN = (P, T, F, M_0, I, O, \varphi, \omega, \Omega)$ em que:

- (P, T, F, M_0) é uma RdP com conjuntos de lugares P, transições T e arcos F, e uma marcação binária inicial M_0. $|P|, |T|, |F| > 0$, sendo $F = F_i \cup F_o$, em que F_i = arcos de entrada e F_o = arcos de saída;
- I é um conjunto de sinais lógicos de entrada tal que $|I|>0$;
- O é um conjunto de sinais lógicos de saída tal que $I \cap O = \emptyset$, $|O|>0$;

- φ um mapeamento associando toda transição t_i pertencente T com uma condição de disparo $\varphi(t_i)$ = função booleana de I.

- ω um mapeamento associando todo lugar p_i pertencente a P com uma saída $\omega(p_i)$ pertencente a $(0, 1, -)^{|O|}$, em que (-) significa "não importa";

- Ω uma função que combina a saída ω de todos os lugares marcados Ω: $M \rightarrow (-, 1, 0, c, r_0, r_1, c_0, c_1, c_{01})^{|O|}$. Esta saída combinada pode ser indefinida (-), um (1), zero (0), contraditória (c), zero ou um redundante (r_0, r_1), uma combinação entre contradição e redundância (c_0, c_1, c_{01}).

A mudança de estado da rede é causada pelo disparo de transições. O disparo de uma transição remove a marca de cada um dos pré-lugares (definidos por F_i) desta transição e adiciona uma marca aos pós-lugares (definidos por F_o) desta transição. No processo de disparo, quatro regras devem ser seguidas (FREY, 2000):

1. Uma transição está habilitada se todos os seus pré-lugares (lugares p_j tais que $(p_j, t_i) \in F_i$) estão marcados, e todos os seus pós-lugares (lugares p_j tais que $(t_i, p_j) \in F_o$) estão desmarcados;

2. Uma transição dispara imediatamente se ela estiver habilitada e se sua condição de disparo for satisfeita (valor 1 para o resultado da função $\varphi(t_i)$);

3. Todas as transições disparáveis e que não estão em conflito com outras transições disparam simultaneamente;

4. O processo de disparo é iterado até que uma marcação estável seja atingida, isto é, até que não haja mais transições disparáveis. Disparos iterados são interpretados como simultâneos. Isto significa que uma mudança nos valores dos sinais de entrada não pode ocorrer durante o processo de disparo.

Alcançada uma marcação estável, os sinais de saída são recalculados aplicando-se a função Ω à marcação. A figura 2.5 apresenta um exemplo de sistema cuja estratégia de controle foi modelada por uma rede do tipo *SIPN*.

Nesta figura, representa-se um sistema comumente encontrado em aplicações industriais (acumulador de ar comprimido conectado a compressores e equipado com sensores de pressão) o qual teve sua estratégia de controle modelada por uma *SIPN* (MERTKE & FREY, 2001). A figura 2.5 mostra uma câmara de ar (acumulador) da qual se pode retirar uma parte do ar comprimido por meio de uma válvula instalada na câmara. Dois sensores binários PS1 e PS2 são usados para monitorar a pressão no interior da câmara. Por meio de dois compressores A e B, a câmara é alimentada com ar comprimido. Os compressores geram um sinal elétrico quando são interrompidos (desligamento automático, falha, interrupção temporária). A estratégia de controle deve atender às especificações a seguir:

1. Se a pressão for maior do que 6,1 bar (PS1 alterna para a posição OFF), nenhum dos compressores entra em ação;
2. Se a pressão for menor do que 6,1 bar e maior do que 5,9 bar (PS1 alterna para a posição ON), apenas um dos compressores entra em ação;
3. Se a pressão for menor do que 5,9 bar (PS2 alterna para a posição ON), e ambos compressores entram em ação;
4. Ambos compressores devem trabalhar alternadamente;
5. Se um compressor é interrompido, o outro deve substituí-lo.

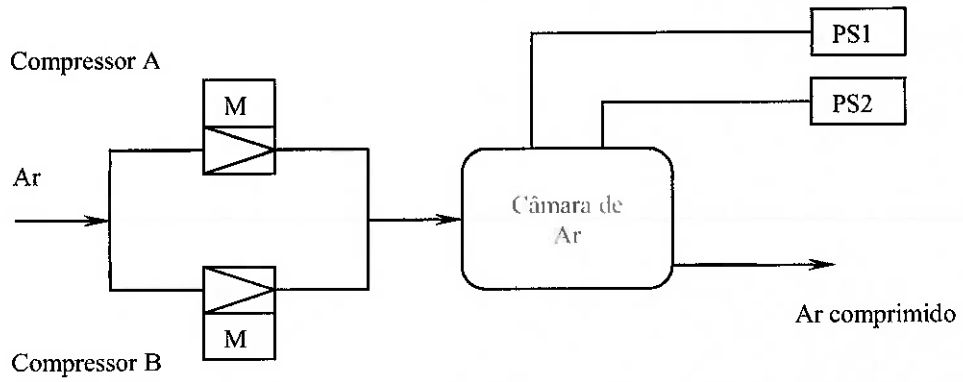


Figura 2.5: Representação do sistema.

Para implementar o controlador, os sinais de I/O são codificados na tabela 2.2. Na figura 2.6 tem-se uma possível solução para o sistema de controle, modelada por uma *SIPN*. Junto a cada lugar da RdP foi escrito o

mapeamento *ω(pi)=(o1,o2)* dos sinais de saída correspondentes, e junto a cada transição, o mapeamento *φ(ti)* correspondente.

Tabela 2.2: Codificação dos sinais do controlador.

Sinais de entrada	Significado do valor valor lógico "1"	Sinais de saída	Significado do valor lógico "1"
i1	Pressão < 6,1 bar	o1	Compressor A em ação
i2	Pressão < 5,9 bar	o2	Compressor B em ação
i3	Compressor A perturbado		
i4	Compressor B perturbado		

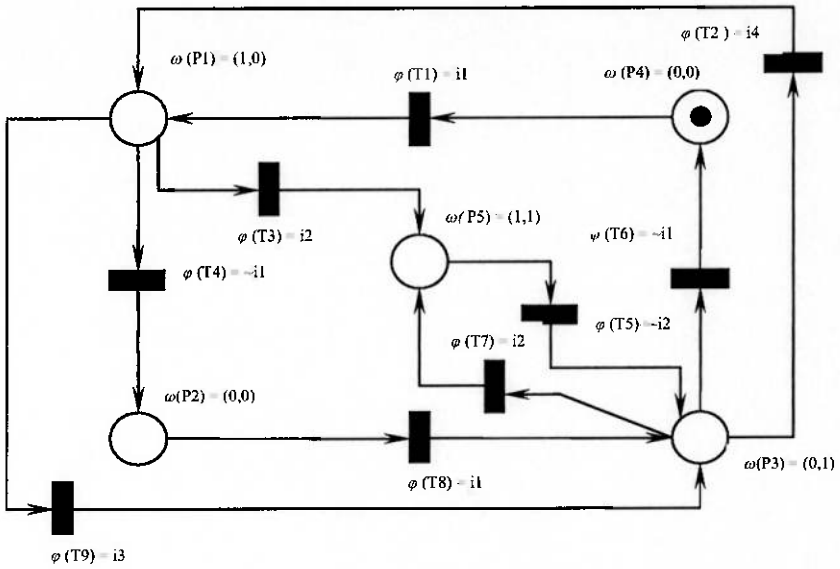


Figura 2.6: Controlador modelado em *SIPN*.

2.4. Controladores programáveis (CPs) e suas linguagens de programação

Um CP é, segundo definição da *NEMA - National Electrical Manufacturers Association*, um aparelho eletrônico digital que utiliza uma memória programável para o armazenamento interno de instruções a fim de implementar funções específicas tais como lógica, seqüenciamento, temporização, contagem e operações aritméticas, para controlar máquinas ou processos através de módulos de entrada / saída analógicos ou digitais

(SOUZA, 2001). A estrutura de um CP pode ser dividida em três partes, conforme ilustrado na figura 2.7.

Figura 2.7: Estrutura simplificada de um CP.

Um CP é formado por cinco elementos básicos: processador, memória, sistema de entrada / saída, fonte de alimentação e terminal de programação. A três partes principais (processador, memória e fonte de alimentação) formam o que se chama de *CPU – Central Processing Unit*. O processador lê dados de entrada de dispositivos de comando e/ou detecção, executa o programa do usuário armazenado na memória e envia dados de saída para comandar os dispositivos de atuação e monitoração. Este processo de leitura das entradas, execução do programa e controle das saídas é feito de uma forma cíclica e é chamado de ciclo de varredura (SOUZA, 2001).

O sistema de entrada / saída de sinais forma a interface pela qual dispositivos de campo (dispositivos de comando, de monitoração, de atuação, de detecção) são conectados ao controlador. O propósito desta interface é condicionar os vários sinais recebidos ou enviados ao mundo externo. Sinais provenientes de dispositivos de comando e de detecção tais como *push-buttons*, chaves limites, sensores analógicos, chaves seletoras e chaves tipo tambor (*thumbwheel*), são conectados aos terminais dos módulos de entrada. Dispositivos de atuação e de monitoração, como válvulas solenóides, lâmpadas-piloto e outros, sãos conectados aos terminais dos módulos de saída. Uma fonte de alimentação fornece a energia necessária para a devida operação do CP e da interface dos módulos de entrada e saída.

Outro componente do CP é o dispositivo de programação. Embora seja considerado como parte do controlador, o terminal de programação, como também é chamado, é requerido apenas para carregar o programa de

aplicação na memória do controlador. Uma vez carregado o programa, o terminal pode ser desconectado do controlador. Atualmente, usa-se o microcomputador com um *software* próprio para programar o CP e devido à capacidade de processamento do mesmo, este também é utilizado na edição e depuração do programa (SOUZA, 2001).

As linguagens comumente utilizadas na programação de CPs e padronizadas pela norma *IEC 61131-3* podem ser divididas em três grupos: textuais, gráficas e tabulares. Nas linguagens textuais, os procedimentos de controle são descritos textualmente por meio de símbolos, letras e expressões matemáticas. Dentre as linguagens textuais, têm-se (MIYAGI, 1996):

- Álgebra Booleana: relações lógicas são representadas por meio de expressões booleanas. A desvantagem deste tipo de linguagem é a incapacidade de representar temporizações e seqüencializações;
- *IL* (*Instruction List*): trata-se de uma lista composta por comandos, os quais correspondem a funções, tais como *LD* (*load*), *AND*, *OR*, ST (*store*); e códigos das entradas e saídas organizados seqüencialmente à ordem em que estes devem ser executados;
- *ST* (*Structured Text*): baseia-se em uma representação em linguagem de alto nível, de forma que a ordem em que o texto é escrito não tem relação com a ordem de execução do programa. A vantagem desta linguagem está na capacidade de estruturação de programas.

A figura 2.8 mostra uma mesma operação lógica descrita nas três linguagens anteriormente citadas.

$$O10 = \bar{I1}.I2.(I3.\bar{I4} + I5.I6).I7 + I8.I9$$

(a) Álgebra booleana.

```
O10:=I1 & NOT I2 & (I3 & NOT I4 OR I5 & I6) &
I7 OR I8 & I9
```

(b) ST.

```
LD      I1
ANDN    I2
AND(    I3
ANDN    I4
OR(     I5
AND     I6
)
)
AND     I7
OR(     I8
AND     I9
)
ST      O10
```

(c) IL.

Figura 2.8: Representações em linguagens textuais de uma operação booleana.

As linguagens gráficas são de fácil visualização e identificação do fluxo de procedimento de controle. Devido à característica gráfica destas linguagens, o projeto, a programação, a depuração e a manutenção de programas de CPs são facilitados, e a ocorrência de erros é dificultada. A seguir descrevem-se sucintamente estas linguagens, de acordo com MIYAGI (1996).

- Diagrama de Relés (*Ladder Diagram - LD*): esta linguagem baseia-se em representações gráficas de relés (*switches*) cujo chaveamento depende de sinais de entrada ou variáveis internas, e de núcleos (enrolamentos) os quais quando excitados por corrente elétrica comportam-se como memórias para armazenagem de valores de variáveis e sinais de saída. A figura 2.9 ilustra um diagrama de relés. O processamento de sinais ao longo das linhas (*rungs*) em *LDs* é feito ciclicamente, de cima para baixo e da esquerda para a direita;

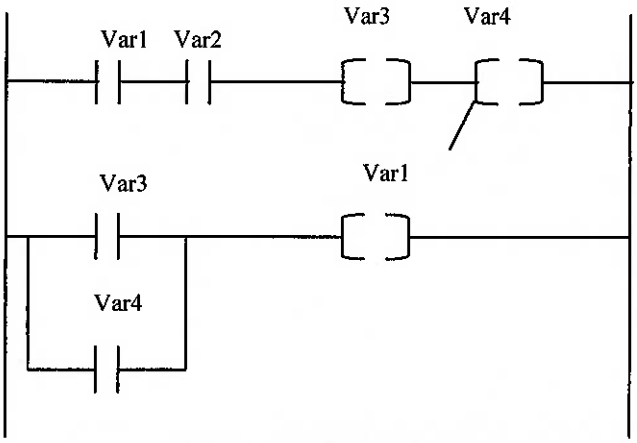


Figura 2.9: Exemplo de um programa escrito em *LD*.

- *FBD* (*Function Block Diagram*): as operações são representadas por meio de blocos lógicos (AND, OR, NOT, etc.) e blocos aritméticos. A figura 2.10 ilustra esta linguagem. Se apenas funções lógicas estiverem presentes, é também chamada de Diagrama de Circuito Lógico.

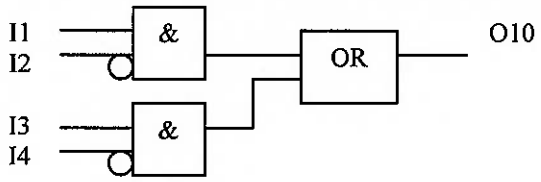


Figura 2.10: *FBD*.

- Fluxograma: trata-se da mesma técnica utilizada no desenvolvimento de algoritmos para computadores, somando-se funções adequadas de controle. É adequada para controles sequenciais;

- *SFC* (*Sequential Flow Chart*): é uma descrição baseada em *steps* (condições) e eventos (transições) que alteram o estado do sistema modelado. É uma linguagem originada da RdP, sendo adequada para o controle de sistemas de manufatura. A figura 2.11 apresenta um programa em *SFC*.

As linguagens tabulares descrevem as operações por meio de tabelas contendo ações correspondentes para cada passo, uma identificação do próximo passo a ser realizado e a condição necessária para que ocorra a transição para o próximo passo.

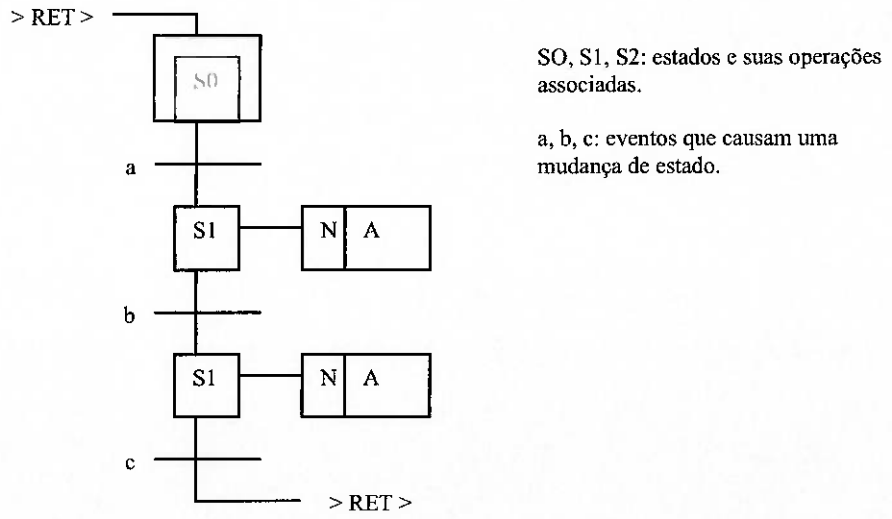


Figura 2.11: Exemplo de um programa em *SFC*.

As normas do *IEC* apresentam o *IL*, *ST*, *LD* e *FBD* como linguagens de programação de CPs (MIYAGI, 1996), ao passo que o *SFC* e os blocos são definidos como elementos compartilháveis e próprios para a representação de programas de CP.

2.5. Extensible Markup Language (XML) e Petri Net Markup Language (PNML)

Segundo DYKES & TITTEL (2005), *XML* é uma notação baseada em *tags* que estabelece uma técnica para identificar, categorizar e organizar informações. As *tags* descrevem documentos, dados e sua organização. O conteúdo destas *tags* envolve as informações (textos, referências para imagens, tabelas) de que um determinado documento trata. A figura 2.12 ilustra um trecho de um documento em notação *XML*.

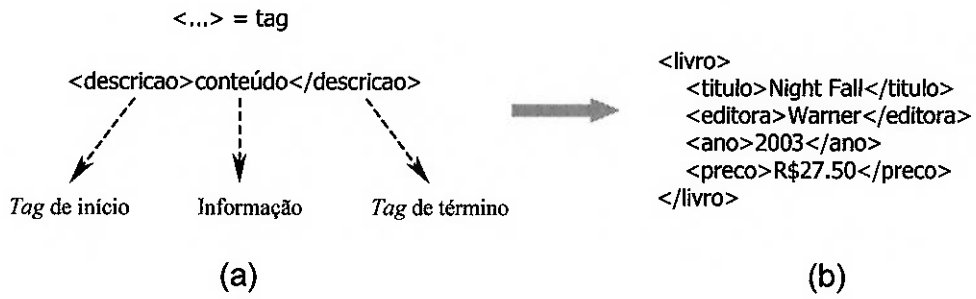
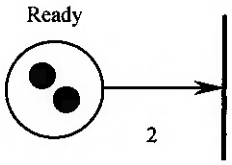


Figura 2.12: (a) Estrutura em *XML*. (b) Trecho de um documento em *XML*.

O conteúdo envolvido por um par início-término de *tags* pode apresentar não só informações relativas à descrição de um documento, mas também pares internos de *tags* com descrições, configurando assim, um conjunto de informações segundo um formato em cascata, conforme visto na figura 2.12.

Fundamentada em *XML*, o *Petri Net Markup Language* (*PNML*) é uma notação destinada ao intercâmbio de modelos em RdP. Com ela, pode-se descrever um modelo em RdP segundo um conjunto pré-determinado de *tags* que relacionam os elementos lugar, transição e arco de uma RdP. Segundo BILLINGTON et al. (2003), o *PNML* foi projetado para ser um formato padrão de intercâmbio, independente de ferramentas (*softwares*) e de plataformas. Em adição, tal formato apresenta características como: flexibilidade, que permite a representação de diversos tipos de RdP; compatibilidade, que permite a definição de diferentes tipos de *tags* para uma perfeita caracterização dos diversos tipos de RdP; ausência de ambigüidade entre modelos e entre tipos de RdP. Como um exemplo, figura 2.13 apresenta uma RdP e sua descrição segundo a notação *PNML*.



(a).

```
<pnml xmls="http://www.example.org/pnml">
  <net id="n1" type=" http://www.example.org/pnml/PTNet">
   <name>
    <text>A Petri Net example</text>
   </name>
   <place id="p1">
    <graphics>
     <position x="20" y="20"/>
    </graphics>
    <name>
     <text>Ready</text>
     <graphics>
      <offset x="-10" y="-8"/>
     </graphics>
    </name>
    <initialMarking>
     <text>2</text>
    </initialMarking>
   </place>
   <transition id="t1">
    <graphics>
     <position x="20" y="20"/>
    </graphics>
    <toolspecific tool="PN4all" version="1.0">
    </toolspecific>
   </transition>
   <arc id="a1" source="p1"> target="t1">
    <graphics>
     <position x="30" y="5"/>
     <position x="60" y="5"/>
    </graphics>
    <inscription>
     <text>2</text>
     <graphics>
      <offset x="15" y="-2">
     </graphics>
    </inscription>
   </arc>
  </net>
</pnml>
```

(b)

Figura 2.13: (a) RdP. (b) Descrição em *PNML*.

Para que um modelo em RdP, descrito em *PNML*, seja interpretado sem ambigüidade, deve-se definir um conjunto de *tags* para cada tipo de RdP, sendo que a cada uma atribui-se um significado específico. O *PNML* não é um trabalho fechado, permitindo assim novas contribuições. A tabela 2.3 apresenta algumas descrições para as *tags* do modelo em *PMNL* da figura 2.13.

Tabela 2.3: Elementos *PNML*.

Tipo	*Tag*	Atributo *XML*	Descrição
Documento *PNML*	<pnml>	-	-
Rede de Petri	<net>	id: ID	Identificador
Place	<place>	id: ID	Identificador
Transition	<transition>	id: ID	Identificador
Arc	<arc>	id: ID	Identificador
		Source: IDRef	Referência para elemento da RdP
		Target: IDRef	Referência para elemento da RdP
Gráfico	<graphics>	-	Posição do elemento na tela

Nome	<name>	-	Nome do elemento da RdP
Ferramenta	<toolspecific>	tool: string	Ferramenta utilizada para leitura do modelo em PNML
		version: string	
Posição	<position>	x: X	Coordenada absoluta no eixo X
		y: Y	Coordenada absoluta no eixo Y
Offset	<offset>	x: X	Coordenada relativa no eixo X
		y: Y	Coordenada relativa no eixo Y
Texto	<texto>	-	Comentário, número, etc
Inscrição	<inscription>	-	Inscrição (peso) de um arco
Marcação inicial	<inicialMarking>	-	Marcação inicial de um lugar

Assim sendo, dada uma classe específica de RdP (i.e., *CE*, LT, *RTPN*, *APN*), pode-se empregar a notação oferecida pelo *PNML* e, caso seja necessário, novas tags podem ser criadas para que se especifique apropriadamente as características do tipo de RdP utilizado na modelagem de um SED.

2.6. *Orientação a `Objetos`*

A orientação a `objetos` é uma abordagem, voltada ao desenvolvimento de *software*, que propõe um enfoque baseado no relacionamento que existe entre o homem, a natureza e as entidades feitas pelo homem. Desta forma, o domínio do problema é caracterizado como um conjunto de `objetos`[2] que têm `atributos` e comportamentos específicos. Os `objetos` são manipulados como uma coleção de funções (chamadas `métodos`, `operações` ou `serviços`) e comunicam-se uns com os outros através de um protocolo de mensagens (PRESSMAN, 2002).

[2] Termos Específicos da Orientação a Objetos são apresentados em `Courier New`.

O elemento básico da orientação a objeto é a classe. Uma classe é um tipo de objeto que pode ser definido pelo programador para descrever uma entidade real ou abstrata. Pode-se entender uma classe como um modelo ou uma especificação para certos objetos, ou seja, a descrição genérica dos objetos individuais pertencentes a determinado conjunto (JANDL JÚNIOR, 2002).

Segundo ANGEL-RESTREPO (2004), um objeto é definido como uma estrutura que possui dados, estados e métodos. Os dados são a especificações da informação que caracterizam o objeto, os métodos são as formas de manipular os dados, permitindo alterar o estado do objeto. Os métodos podem ser de três tipos: métodos de construção / destruição, que permitem que um objeto passe a ser parte do sistema ou deixe de ser; métodos com acesso a escrita, cuja função é modificar os dados do objeto; e métodos com acesso a leitura, que permitem o acesso aos dados do objeto, sem alterá-los.

No paradigma de orientação a objetos, é necessário observar algumas características intrínsecas (ANGEL-RESTREPO, 2004):

- Classificação: é um mapeamento entre objetos e classes conhecidas. Se um objeto não se "encaixa" em nenhuma das classes conhecidas, cria-se uma nova classe;
- Reutilização: consiste em utilizar instâncias de uma classe fazendo referência aos seus métodos de criação. Tal característica proporciona a possibilidade de construir diversas entidades que façam uso da mesma classe de objetos;
- Herança: consiste no fato de que uma classe contém as propriedades de todas as classes de objetos das quais ela é descendente na árvore de classificação das classes;

- Composição: propriedade de um objeto poder ser composto de outros objetos ou construído com base neles, permitindo a construção de objetos complexos a partir de objetos relativamente simples;
- Encapsulamento: combinação de dados e procedimentos que manipulam os dados em uma única classe. O acesso ao objeto fica limitado a uma interface controlada, de forma que o dado seja protegido;
- Polimorfismo: capacidade de um objeto ter diversos métodos com o mesmo nome, porém com formas diferentes, isto é, com número e / ou tipo de parâmetros diferentes.

2.7. *Análise de Requisitos*

Uma etapa indispensável para projeto de *software* é a "análise de requisitos". Segundo PRESSMAN (2002), a análise de requisitos é uma tarefa de engenharia de *software* que "vence o espaço" entre a engenharia de requisitos, no nível de sistemas, e o projeto de *software*. As atividades de engenharia de requisitos resultam na especificação das características operacionais do *software* (função, dados e comportamento), indicam a interface do *software* com outros elementos do sistema e estabelecem restrições a que o *software* deve atender. A análise de requisitos fornece ao projetista de *software* uma representação da informação, função e comportamento, que podem ser traduzidos para os projetos dos dados, arquitetura e interface e para o nível de classes.

A análise de requisitos de *software* pode ser dividida em cinco áreas de esforço: reconhecimento do problema, avaliação e síntese, modelagem, especificação e revisão. A meta é reconhecer os elementos básicos do problema tal como são percebidos pelos clientes / usuários do *software* a ser desenvolvido (PRESSMAN, 2002). Esta etapa garante o pleno reconhecimento do problema.

O analista deve definir todos os objetos de dados observáveis externamente, avaliar o fluxo e o conteúdo de informação, definir e refinar todas as funções do *software*, entender o comportamento do *software* no contexto dos eventos que afetam o sistema, estabelecer as características das interfaces do sistema e descobrir restrições adicionais de projeto. Cada uma destas tarefas serve para descrever o problema de modo que uma abordagem ou solução global possa ser sintetizada.

Durante a avaliação e síntese de solução, o principal foco do analista é "o que" e não "como". A preocupação se baseia em quais dados o sistema produz e consome, quais funções o sistema deve desempenhar, que comportamentos o sistema exibe, que interfaces são definidas e que restrições se aplicam.

Para que as metas da análise de requisitos sejam atingidas, convém ao analista a utilização de técnicas e ferramentas desenvolvidas para este fim. A seguir é discutida uma ferramenta de ampla utilização na engenharia de *software*, os "casos de uso".

2.8. Casos de Uso

Segundo COCKBURN (1999), um caso de uso é uma descrição das possíveis seqüências de interações entre o sistema em discussão e seus atores externos, relacionados a um objetivo particular.

À medida que os requisitos são elicitados nas etapas iniciais da análise de requisitos, o engenheiro de *software* pode criar um conjunto de cenários que identifica uma linha de uso para o sistema a ser construído. Estes cenários são chamados de casos de uso, e fornecem uma descrição de como o sistema será usado (PRESSMAN, 2002).

A primeira etapa na criação de um caso de uso é a definição do ator. Um ator representa cada diferente tipo de pessoas (ou dispositivos) que usam o sistema ou o produto. Como a elicitação dos requisitos é uma atividade

evolutiva, nem todos os atores são identificados durante a primeira iteração. É possível identificar atores secundários quando se toma maior conhecimento do sistema.

Definidos os atores, os casos de uso podem ser desenvolvidos tomando como base os modos que estes atores interagem com o sistema. Algumas questões que esta etapa deve responder são quais tarefas ou funções principais são desempenhadas pelo ator; quais informações do sistema o ator vai adquirir, produzir ou modificar; quais informações externas o ator deve informar ao sistema; e se o ator deseja ser informado de modificações inesperadas.

Para PRESSMAN (2002), cada caso de uso fornece um cenário não-ambíguo de interação entre um ator e o *software*. Ele pode também ser usado para especificar requisitos de tempo ou outras restrições do cenário.

2.9. Qualidade de Software

Qualidade pode ser definida e medida se uma melhoria é atingida. Entretanto, o maior problema da qualidade em engenharia e gerenciamento é que o termo qualidade é ambíguo, e comumente mal entendido (KAN, 2002).

No desenvolvimento de *software*, a qualidade do projeto abrange os requisitos, as especificações e o projeto do sistema. A qualidade da conformidade é um assunto concernente, principalmente à implementação. Se a implementação segue o projeto e o sistema resultante satisfaz os requisitos e metas de desempenho, a qualidade de conformação é alta (PRESSMAN, 2002).

Sendo assim, pode-se dizer que um *software* é de qualidade quando concilia qualidade de projeto e de conformidade. A satisfação do usuário pode ser entendida como uma relação intuitiva, representada abaixo (PRESSMAN, 2002):

"Satisfação do usuário = produto adequado + máxima qualidade +

entrega dentro do orçamento e do cronograma"

Resumidamente, a qualidade de *software* pode ser definida formalmente como conformidade com requisitos funcionais e de desempenho explicitamente declarados, padrões de desenvolvimento explicitamente documentados e características implícitas, que são esperadas em todo *software* desenvolvido profissionalmente.

2.10. Teste de Software

Uma vez gerado o código-fonte, o *software* deve ser testado para descobrir e corrigir tantos erros quanto possíveis antes de ser entregue ao seu cliente. O teste de *software* é um elemento crítico da garantia de qualidade de *software* e representa a revisão final da especificação, projeto e geração de código.

Segundo MYERS (1979), algumas regras podem servir como objetivo do teste:

- Teste é um processo de execução de um programa com a finalidade de encontrar um erro.
- Um "bom" caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.
- Um teste "bem" sucedido é aquele que descobre um erro ainda não descoberto.

Para que se tenha um conhecimento pleno de projeto de casos de testes efetivos, é necessário compreender os princípios básicos que guiam o teste de *software*. DAVIS (1995) sugere um conjunto de princípios de teste:

- Todos os testes devem ser relacionados aos requisitos do cliente;
- Os testes devem ser planejados muito antes do início do teste;

- O Princípio de Pareto se aplica ao teste de *software* (isso implica que 80% de todos os erros descobertos no teste estão relacionados a 20% de todos os componentes do programa);
- O teste deve começar em componentes individuais e se estender aos conjuntos integrados de componentes e finalmente ao sistema;
- Teste completo não é possível, pois a permutação de caminhos é excepcionalmente grande, impossibilitando percorrer todas as combinações.

3. FERRAMENTA PARA MODELAGEM, SIMULAÇÃO, PROGRAMAÇÃO E SUPERVISÃO DE SISTEMAS DE AUTOMAÇÃO

3.1. Escolha das linguagens de programação

3.1.1. Linguagem para a implementação da ferramenta

Como técnica de modelagem de *software* empregada para o desenvolvimento da ferramenta, foi adotada a orientação a `objetos`. Com ela podem-se construir aplicações que facilitam o reuso de código, dada a existência de mecanismos como `herança` e `polimorfismo`. Além disso, a orientação a `objetos` possibilita a modelagem adequada de estruturas de dados complexas e também dos relacionamentos entre estas estruturas. Por último, uma grande comunidade de programadores tem adotado a modelagem orientada a `objetos` como uma técnica de comprovada eficiência no desenvolvimento de *software* (MUNDOOO, 2006).

Sendo assim, dentre as linguagens disponíveis para a implementação da ferramenta, optou-se por avaliar apenas aquelas que suportam o paradigma de orientação a `objetos`. As linguagens mais conhecidas são C++, C#, Java, Eiffel e *SmallTalk*. São difundidas e consideradas adequadas para o desenvolvimento do *software* (MUNDOOO, 2006).

CULWIN (1997) relatou uma pesquisa na qual analisa as informações de grandes desenvolvedores de *software*, e compilou os dados em uma tabela comparativa. Os colaboradores foram instruídos para qualificar as características das linguagens com notas de -5 a 5, deixando em branco os campos que não obtivessem o conhecimento necessário para responder. Os resultados são mostrados na tabela 3.1.

Tabela 3.1: Comparativo entre linguagens orientadas a objeto (Adaptado de CULWIN, 1997).

Quesito	Java	C++	Ada '95	Smalltalk	Eiffel
Polimorfismo	3.41	3.38	3.36	3.90	3.25
Genéricos	3.47	2.83	4.40	2.62	3.86
Encapsulamento	3.71	3.76	4.25	4.30	4.75
Herança	3.61	3.10	2.58	3.95	3.11
Reuso	3.53	2.35	3.33	4.19	3.86
Simplicidade	3.48	-2.24	-0.35	2.95	2.44
Segurança de tipos	3.65	-0.03	4.36	1.11	3.50
Gerenciamento de Memória	3.66	-1.42	0.05	3.65	3.50
Interface e Implementação	2.95	1.80	4.08	2.58	3.22
Média das Notas	3.41	1.49	2.97	3.26	3.47
Número de colaboradores	43	66	28	21	9

Os dados da tabela 3.1 indicam que, considerando o número de respostas (número de pessoas com conhecimento sobre a linguagem) e a nota média obtida, a linguagem Java se destaca das demais sendo, portanto, a linguagem escolhida para a implementação da ferramenta.

Ao avaliar e selecionar uma linguagem para o projeto da ferramenta, que permite a edição e a análise de modelos em RdP, torna-se necessário definir alguns critérios de comparação. Para uma ferramenta de apoio ao projeto de automação residencial, é interessante ressaltar que algumas características são convenientes. Uma fácil adaptação a sistemas operacionais de distribuição livre diminui o custo de implementação da solução de automação. Um *software* portável garante que uma mesma implementação em diferentes sistemas operacionais seja possível. Ferramentas de desenvolvimento de disponibilidade livre e sem custos também são atrativos para o projetista. Facilidade e

disponibilidade de documentação e bibliotecas de códigos, bem como uma comunidade receptiva e prestativa facilitam o desenvolvimento do *software*.

A linguagem Java atende a todos estes requisitos. O *software* de edição e análise deve neste caso ser acionado através de uma máquina virtual (*JVM - Java Virtual Machine*), que funciona como uma ponte entre a linguagem Java e o sistema operacional. Esta *JVM* está disponível para os principais sistemas operacionais do mercado e remete a um *software* de "boa" portabilidade. Os compiladores, as ferramentas de desenvolvimento e a *JVM* são disponibilizadas de forma livre, o que torna o projeto livre de custos relativos à utilização de *softwares* pagos.

Considerando estes pontos, para o desenvolvimento do presente trabalho, está sendo utilizada a versão 5 do *JSE* (*Java Standard Edition*), que contém o compilador Java e a *JVM* (máquina virtual desenvolvida pela *Sun Microsystems*, 2006), ferramentas imprescindíveis para o desenvolvimento de aplicações em Java. Como ambiente de desenvolvimento, no qual arquivos-fonte podem ser implementados considerou-se o *software Netbeans* 5.0, que facilita a geração de código e de processos de depuração (NETBEANS, 2006).

3.1.2. Linguagem de programação do CP

Em uma proposta apresentada por FREY & MINAS (2001), a modelagem de estratégias de controle para CPs é feita inicialmente de forma gráfica utilizando-se *SIPNs*. Em seguida, desenvolvendo-se algoritmos adequados, pode-se converter automaticamente a estratégia de controle para uma linguagem apropriada ao uso em CPs, a Lista de Instruções (*IL – Instruction List*, também conhecida como *Statement List – STL*).

A dinâmica de um programa de controle de um SED modelado por uma *SIPN* assemelha-se à forma de trabalho de um CP devido ao fato de que em ambos os eventos e as alterações de estado são governados por sinais de entrada e sinais de saída. Com relação à linguagem *IL*, MERTKE & FREY (2001) afirmam ser possível utilizá-la para implementar programas de controle

modelados em RdP, sem desconsiderar características fundamentais de SEDs como concorrência, assincronismo e sincronismo, não-determismo.

Como o objetivo do projeto em questão engloba também o desenvolvimento de um ambiente gráfico para programação de CPs, baseado em RdP, a abordagem utilizada neste projeto baseia-se na proposta de FREY & MINAS (2001). Outros trabalhos encontrados na literatura também abordam o uso de grafos bipartidos, tal como uma RdP, para o projeto de programas de controle que podem ser carregados e executados em CPs após a conversão para um formato apropriado. Um exemplo disso pode ser encontrado em ZHOU & VENKATESH (1998).

Como nas *SIPNs* descritas em FREY & MINAS (2001) é utilizada a linguagem *IL* para a implementação de programas de CPs, isto determina, por sua vez, a linguagem utilizada no projeto para a qual programas de controle executados por CPs e modelados por *SIPNs* serão traduzidos.

3.2. Levantamento de Requisitos

Para que a implementação e o levantamento de requisitos sejam mais eficientes, o desenvolvimento foi dividido em duas etapas. A primeira, relacionada com o algoritmo em si, corresponde ao desenvolvimento da estrutura de dados e ao algoritmo que implementa as regras de execução de RdP. A segunda, relacionada com a interação com usuário, envolve o desenvolvimento de uma interface gráfica para a modelagem e visualização da RdP.

Desta forma, as informações da RdP devem ser passadas do usuário para a interface gráfica, e esta deverá fazer a devida comunicação com o algoritmo, para que o desenvolvimento e a análise do SED modelado pela RdP sejam efetivados.

3.2.1. Requisitos dos Algoritmos

Para que o algoritmo seja capaz de permitir ao usuário a edição e a análise de RdP, ele deve satisfazer algumas funcionalidades (casos de uso):

- Implementação dos elementos da RdP em `objetos`;
- Implementação das regras de execução da RdP;
- Construção de RdP em estrutura de dados;
- Recuperação e edição de RdPs já criadas;
- Definição do estado da RdP (posicionamento das marcas);
- Acesso ao estado da rede de Petri (visualização das marcas).

Estas funcionalidades devem ser garantidas para que a interface seja capaz de traduzir as informações que deverão ser fornecidas para o usuário, ou que o usuário deverá fornecer para a ferramenta. Desta forma, considera-se que o ator destes casos de uso é a interface com o usuário.

3.2.2. Requisitos da Interface

O usuário deve ser capaz de transferir para a ferramenta toda a informação necessária para o funcionamento desta. Para isso, devem-se considerar as seguintes funcionalidades para uma RdP (casos de uso):

- Desenho;
- Edição (alteração de características);
- Visualização;
- Salvamento e recuperação;
- Facilidades de organização (quadros e textos);
- Execução em modo automático, ou passo-a-passo.

Neste caso, o ator destes casos de uso é o usuário final. Ele deve ser capaz de realizar todas estas ações, de forma a satisfazer as necessidades do da ferramenta baseada em RdP.

3.3. Estruturas de dados

3.3.1. Modelagem da RdP segundo a orientação a objeto

Como primeiro passo, foi definido um conjunto de classes chamado "core" (núcleo). Este conjunto contém toda a infra-estrutura necessária para a execução da RdP.

Este conjunto de classes deve realizar a edição de uma RdP e a execução das regras de disparo das transições. Para isso o conjunto deve oferecer a estrutura de dados necessária, bem como os métodos com que a comunicação com os objetos do conjunto é realizada.

Segundo JANDL (2003), um componente pode ser entendido, no seu sentido mais genérico, como um bloco de códigos utilizado em tempo de execução, que possui uma interface definida claramente, por meio da qual provê serviços a outras classes.

No contexto da orientação a objetos, os elementos da RdP (lugar, transição e arco) podem ser tratados como um novo tipo de dado que envolve um conjunto de atributos. Esses elementos carregam consigo informações que os caracterizam no contexto de uma RdP. Como exemplo, tem-se a capacidade e a marcação inicial, informações características e fundamentais do elemento lugar. Com uma transição o mesmo se passa, pois esta pode ser determinística (i.e., estando ativada, dispara após um intervalo de tempo pré-determinado) ou instantânea (i.e., dispara assim que se encontra ativada), ou ainda podem apresentar outras informações como um nome específico. Igualmente com relação ao arco, é necessário saber seu peso, qual lugar e qual transição ele conecta, e seu sentido também. Todos esses aspectos, cada um inerente a um determinado elemento da RdP, precisam ser armazenados para

recuperação e manipulação. Conseqüentemente, lugares, transições e arcos não podem ser analisados simplesmente como variáveis primitivas tal como um número inteiro ou um caractere, mas sim como estruturas complexas constituídas por diversos atributos, o que permite modelá-los como objetos pertencentes a classes.

O relacionamento entre os elementos da RdP, após modelados como objetos, pode ser implementado por meio de métodos[3]. Na figura 3.1 é apresentado o relacionamento entre as classes implementadas na ferramenta e que modelam os elementos da RdP. A notação apresentada nesta figura baseia-se nos diagramas de classes da *UML*[4] (*Unified Modeling Language*).

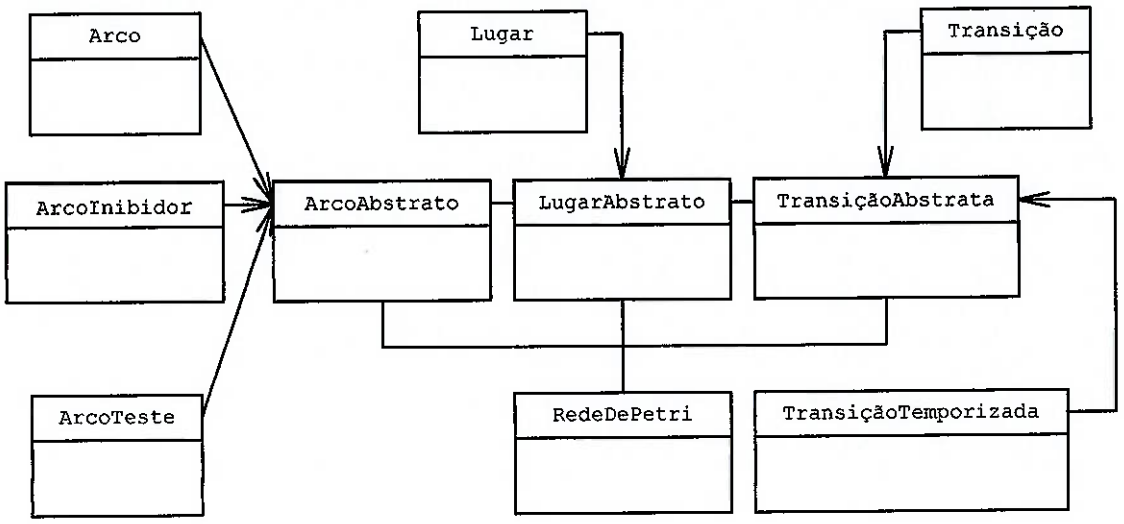


Figura 3.1: Diagrama *UML* do pacote "core".

Na figura 3.1, os relacionamentos entre as entidades são indicados por meio de setas e de linhas. Uma seta conota uma relação de herança, em que a classe conectada na extremidade de onde o arco parte "herda" os atributos e métodos da classe na qual o arco chega. Como exemplo, tem-se a classe ArcoInibidor que herda os métodos e atributos da classe ArcoAbstrato. Já as linhas expressam apenas um relacionamento

[3] Um método, segundo HORTON (2003), pode ser entendido como uma mensagem enviada a um objeto. Mensagem esta que invoca uma ação específica para objeto.
[4] A *UML* é, segundo HORTON (2003), uma notação destinada à modelagem de classes de objetos.

entre classes, definido pela forma como o sistema foi modelado e implementado.

3.3.2. Estrutura dos modelos em RdP descritos segundo a notação *XML*

A ferramenta desenvolvida neste trabalho deve ser capaz de interpretar modelos em RdP descritos segundo a notação *XML*, a qual é considerada pela comunidade de RdP como um formato próprio para intercâmbio de modelos entre *softwares* que trabalhem com RdP (BILLINGTON et al., 2003).

A princípio, optou-se pelo uso da notação *PNML* para a descrição da RdP representativa de um programa de controle de um CP, devido à flexibilidade apresentada por esta notação. No entanto, observou-se que esta não suportava a descrição de estratégias de controle de SMs modeladas por *SIPN*. Desta forma foi necessário criar um novo conjunto de *tags*, baseado no *PNML*, de modo a dar suporte às *SIPNs*.

Ao longo do processamento do arquivo *XML* contendo um modelo em RdP, o *software* interpreta e armazena as informações a respeito de cada lugar, transição e arco existentes neste modelo. Este arquivo descreve o relacionamento, tais como conexões entre os elementos da RdP, marcação inicial da rede, posição na tela de cada um dos elementos e informações textuais, por meio de pares de *tags*.

3.4. *Algoritmo "jogador de marcas"*

O disparo interativo de transições e as alterações resultantes nas marcações em uma RdP é chamado de "jogo de marcas". Implementando-se apropriadamente os elementos da RdP segundo as técnicas da orientação a objetos e também baseando-se nas ferramentas e *APIs* já existentes na linguagem Java, é possível implementar um algoritmo capaz de realizar o "jogo de marcas" (ZHURUWASKI & ZHOU, 1994).

Muitos *softwares* utilizados para modelagem e simulação de SEDs empregam um mesmo algoritmo jogador de marcas. Desta forma, pesquisando-se na literatura, implementou-se um algoritmo "jogador de marcas" similar ao apresentado em ZHURUWASKI & ZHOU (1994). A figura 3.2 mostra o algoritmo desenvolvido.

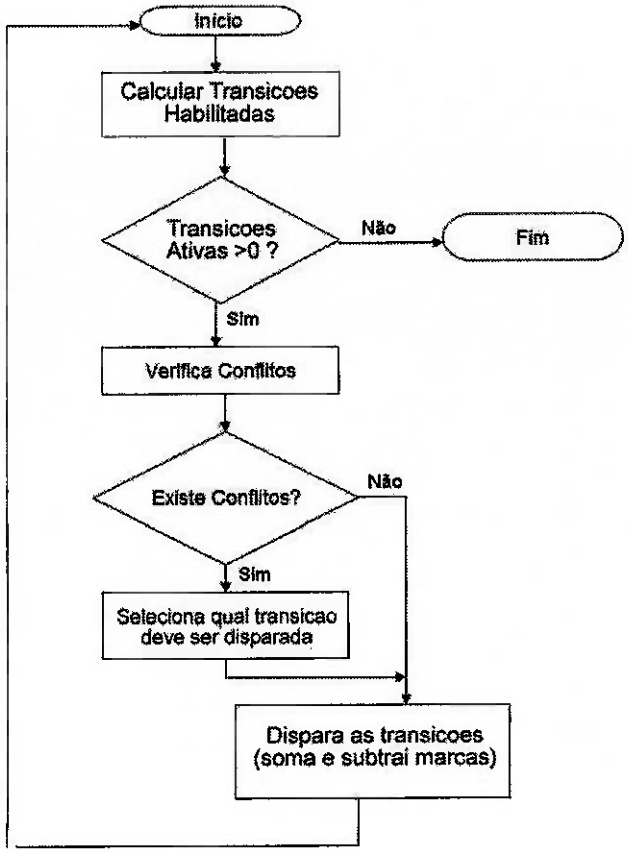


Figura 3.2: Algoritmo para o "jogador de marcas" de uma RdP.

Transições habilitadas

Nesta etapa, o algoritmo procura na RdP por transições que estejam prontas para serem disparadas. Ele gera, então, uma lista com estas transições.

Solucionador de conflitos

Um conflito, em um SED, resulta de uma situação em que um determinado elemento é compartilhado por um ou mais elementos do sistema modelado. Em termos de RdP, duas transições de uma rede estão em conflito

uma com a outra se ambas estão habilitadas e o disparo de uma torna a outra não habilitada (MIYAGI, 1996). Um exemplo, em um SED, seria um robô servindo duas máquinas que demandam algum tipo de serviço prestado apenas por este robô.

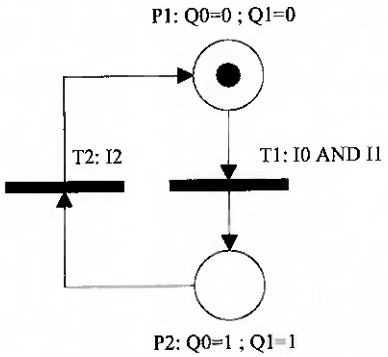
A solução para um conflito, no caso do "jogador de marcas" deste projeto, foi implementada baseando-se em uma escolha aleatória da transição que será disparada quando a RdP for do tipo CE ou LT e, para *SIPN*s, o conflito é automaticamente resolvido pela condição de disparo da transição. Ou seja, dada um estado em que duas transições habilitadas apresentam pré-lugares em comum, será disparada aquela que tiver sua condição de disparo satisfeita primeiro.

Atualização da marcação

Após a ocorrência de um disparo de uma ou mais transições, a RdP apresenta uma nova marcação e, conseqüentemente, o modelo em RdP atinge um novo estado. As marcas são redistribuídas nos lugares da rede.

3.5. Geração de programas de controle de CPs a partir de SIPN

Como resultado da conversão de um modelo de um sistema, representado por um *SIPN*, a ferramenta de conversão produzirá um arquivo contendo um programa para CP em *IL*. A figura 3.3 apresenta um modelo em *SIPN* e o arquivo de saída gerado pela ferramenta para este modelo.



(a) Modelo em *SIPN*.

```
PROGRAM Main

VAR
        PV_1      : BOOL := TRUE ;   (* P1 *)
        PV_2      : BOOL := FALSE ;  (* P2 *)
        Stab      : BOOL:=TRUE:
        (*Stability variable*)
END_VAR

VAR_GLOBAL
        button_pressed          at %IX0.0:BOOL;
        machine_off             at %IX0.1:BOOL;
        time_off                at %IX0.2:BOOL;
        machine_on              at %QX0.0:
        BOOL;
        alert_light at %QX0.1:          BOOL;
END_VAR
```

```
(****Set stability variable  to true*****)
I_0:    S Stab
(******  Transition T1 ******)
I_1:    LD      PV_1    (* pre place P1
*)
        ANDN    PV_2    (* post place
P2 *)
        JMPCN   I_2
        AND button_pressed
      AND machine_on
        JMPCN   I_2
        R       PV_1    (* pre place P1
*)
        S       PV_2    (* post place
P2 *)
        R       Stab
(******  Transition T2 ******)
I_2:    LD      PV_2    (* pre place P2
*)
        ANDN    PV_1    (* post place
P1 *)
        JMPCN   I_3
        AND time_off
        JMPCN   I_3
        R       PV_2    (* pre place P2
*)
        S       PV_1    (* post place
P1 *)
      R Stab
(*****Stability check*******)
I_3:    LD      Stab
        JMPCN I_0
(******** Place P1 ********)
I_4:    LD      PV_1
        JMPCN   I_5
        R machine_on
        R alert_light
(******** Place P2 ********)
```

(b) Programa em *IL*.

Figura 3.3: Modelo em *SIPN* e código em *IL* gerado para o modelo.

Os programas em *IL* gerados pela ferramenta de conversão apresentam a estrutura descrita a seguir.

Declaração de variáveis

Uma variável booleana é declarada para cada lugar do modelo em *SIPN*. Seu valor inicial depende da presença de uma marca no lugar, ou seja, a um lugar com uma marca atribui-se uma variável binária com valor lógico 1. Caso não haja marca, atribui-se o valor lógico 0.

Código para as transições

A compilação de uma transição deve testar se esta se encontra habilitada e se a condição de disparo associada a ela está satisfeita. Se o resultado dos testes indicar um valor lógico 1, conclui-se que todas as condições foram satisfeitas, então a transição dispara. Do contrário, um salto condicional para a próxima transição evita o disparo. O disparo retira a marca de cada um dos pré-lugares e coloca uma marca em cada um dos pós-lugares da transição. Para otimizar o código, a condição de disparo não é avaliada se a transição não estiver habilitada.

Considerando que o processo de disparo é iterativo, sendo interrompido apenas quando não há mais transições disparáveis dada uma configuração de sinais de entrada do CP, uma variável indicadora foi introduzida (Stab). A ela é atribuído o valor 1 no começo do processo (I_0: S Stab) e, caso alguma transição dispare, é atribuído o valor 0. Após a verificação da existência de transições, a variável de parada é testada. Se seu valor for 0, conclui-se que alguma transição disparou e, desta forma, o processo de disparo é iterado. Caso seu valor continue sendo 1, o programa executa um salto condicional para o processamento do código relativo aos lugares. O emprego dessa variável na lógica do programa em *IL* tem a finalidade de modelar a ocorrência de disparos simultâneos de transições, ou seja, recebidos sinais de entrada, deve-se disparar todas as transições habilitadas que tenham suas condições de disparo satisfeitas.

Como exemplo para que se entenda o funcionamento do processo de disparo de uma transição, considere o trecho de código para a transição t1, na figura 3.3 (linha I_1 do programa). Nele, verifica-se se a transição está habilitada por meio das instruções:

```
LD      PV_1
ANDN    PV_2
```

Caso as transições estejam habilitadas, o salto condicional "JMPCN I_2" não é executado. Se as transições estiverem habilitadas, o programa executa as instruções:

AND button_pressed

AND machine_on

Caso o resultado desta verificação seja verdadeiro, o programa interpreta a condição de disparo como satisfeita, o segundo salto condicional "JMPCN l_2" não é executado e, a atualização do estado da RdP é feita com as instruções:

R PV_2

S PV_2

Por fim, o valor da variável de parada é resetado por meio da instrução:

S Stab

E o processo de iteração continua até que não haja mais transições disparáveis.

Código para os lugares

Se um lugar contém uma marca, então a função de saída correspondente deste lugar é executada, isto é, sinais de saída são gerados. Se o lugar não contém uma marca, então um salto condicional para o próximo rótulo[5] no programa é executado e o segmento de código relativo à função de saída não é executado. Nota-se que a implementação da função de saída resulta em um valor 1 ou 0 (verdadeiro ou falso) para cada um dos sinais de saída do modelo em *SIPN*. Para um valor de saída indefinido, a execução do programa do CP permanece no último valor de saída válido, ou seja, anterior ao indefinido.

Como exemplo de execução do código relativo um lugar, considere novamente na figura 3.3(b) o trecho de código iniciado no rótulo l_4. Nele, verifica-se se a variável PV_1 (relativa ao lugar P1) contém o valor 1, isto é, se há uma marca neste lugar. Caso esta condição resulte em verdadeiro, o salto

[5] As identificações l_0, l_1, l_2, ... , l_3 na Fig.12(b) são os rótulos que sinalizam ao programa o início de um novo bloco de instruções.

condicional "JMPCN I_5" não é realizado, e o programa executa as instruções relativas aos sinais de saída do CP:

R machine_on

R alert_light

Se não houver a marca, ou seja, PV_1 tem valor 0, executa-se o salto condicional e os sinais de saída não são gerados. A instrução "RET" no rótulo I_6 do programa determina o retorno ao início do programa principal, assim, o programa é executado continuamente.

3.6. Desenvolvimento da Interface Gráfica

Segundo BASS (1993), a interface com usuário de um sistema é um dos primeiros determinantes da satisfação do usuário com o sistema. O desenvolvedor de um sistema interativo deve estar concentrado nos parâmetros de ciclo de vida e usabilidade. Usabilidade é um parâmetro de tempo de execução, importante quando o usuário está realmente utilizando o sistema. O ciclo de vida começa na concepção do sistema e termina quando o sistema não está mais sendo usado.

ERICKSON (1990) diz que problemas de interface são freqüentemente óbvios. Já as soluções, são menos óbvias. Pode ser difícil achar uma solução que resolva um problema particular sem criar novos problemas. Mesmo assim, uma solução separada para cada problema pode resultar numa interface com tanta complexidade que deixa de ser usável. O que é realmente necessário é uma solução que "elegantemente" resolva uma série de problemas.

Do ponto de vista de projetistas, o desenvolvimento de uma interface sofre das mesmas dificuldades encontradas no desenvolvimento de *software* de propósito geral, além de outros problemas como: grande disparidade de conhecimento geral e específico entre usuários, influência do estado emocional no desempenho de uma tarefa, influência de fatores humanos de difícil

avaliação, ausência de padronização de métodos e recursos interativos (AGUILERA FERNANDES, 1993).

Levando-se em consideração os problemas e as dificuldades apresentadas, e os requisitos levantados anteriormente, o projeto e desenvolvimento de uma interface com usuário podem ser iniciados.

Utilizando ainda o paradigma de orientação a objetos, dentro da linguagem Java, é possível desenvolver componentes gráficos que implementem as funcionalidades exigidas no levantamento de requisitos.

Desta forma, o projeto utilizou primitivas gráficas e componentes de bibliotecas padrões da linguagem Java para criar um elemento gráfico pelo que é possível a edição e visualização da RdP. Este componente foi chamado de PetriPanel.

O PetriPanel é capaz de receber eventos do mouse (como clique, movimentação do cursor e arrasto) e criar a RdP através das informações inseridas pelo usuário. Na figura 3.4 é possível visualizar a ferramenta em execução.

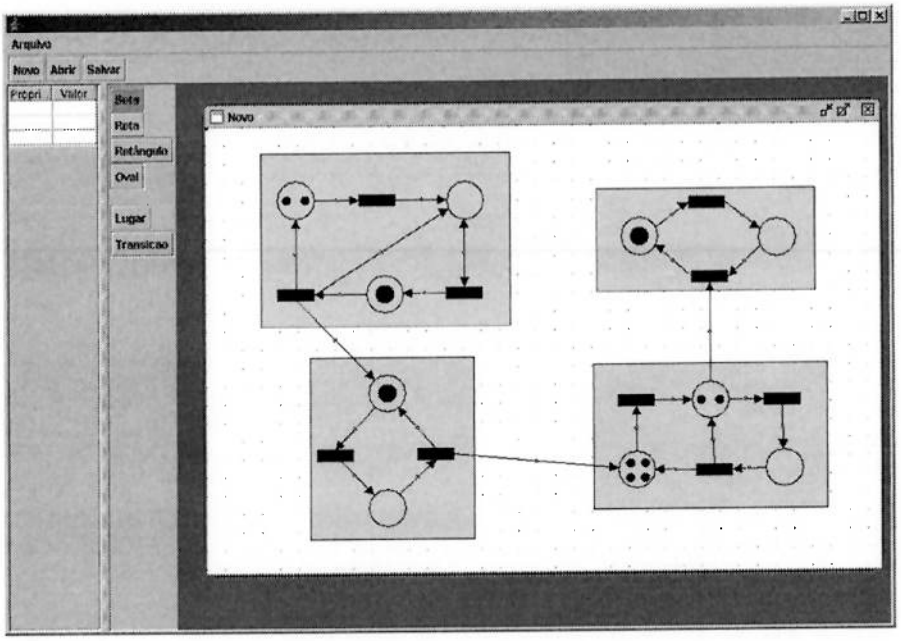


Figura 3.4: Ferramenta em execução.

Através da janela da ferramenta é também possível salvar e recuperar a RdP no sistema de arquivos do sistema operacional. É possível também executar a rede e visualizar o seu estado em tempo de execução.

Além disto, existem botões e janelas para manipulação de variáveis, caso da *SIPN*, bem como ferramentas para a monitoração e modificação dos sinais declarados.

3.7. Testes

Segundo PRESSMAN (2002), o teste é um processo de execução de um programa computacional com a finalidade de encontrar um erro. Portanto, para validar a implementação dos algoritmos propostos, é necessário realizar alguns testes.

3.7.1. Criação e edição de RdPs

ram desenvolvidas algumas RdP para serem executadas pelo conjunto "core".

Primeiramente, o algoritmo foi testado com RdP relativamente simples, como a mostrada na figura 3.5.

Figura 3.5: RdP simples.

Após esta primeira fase de testes, o algoritmo foi testado com RdP de maior complexidade.

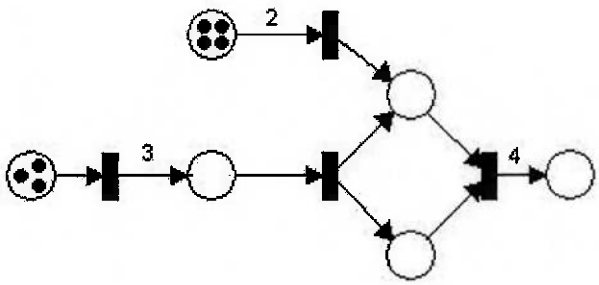


Figura 3.6: RdP com conflito e arcos com peso.

Na figura 3.6, tem-se uma RdP com arcos com peso, com lugares com mais que uma marca e transições em conflitos potenciais.

Foram utilizadas ainda como teste duas RdPs que modelam atividades num sistema de automação. A primeira modela um botão que liga uma lâmpada caso esteja desligada, e desliga a lâmpada, caso esteja ligada. A segunda modela um caso de um sensor de presença, que acende uma lâmpada quando é acionado, aguarda um tempo e desliga.

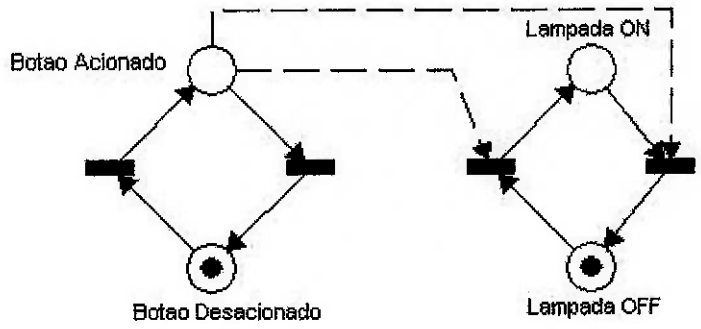


Figura 3.7: RdP que modela um botão liga / desliga.

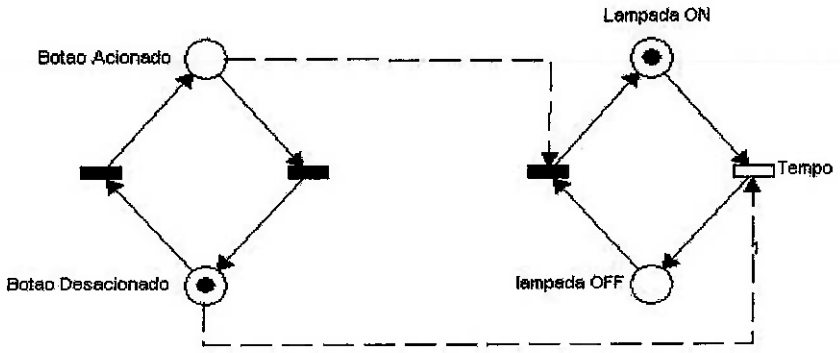
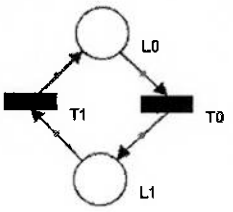


Figura 3.8: RdP que modela um sensor de presença.

3.7.2. Descrição da RdP em *XML*

Para os testes de conversão entre a estrutura de dados da rede de Petri em orientação a `objeto` e *XML*, foram utilizadas duas redes como exemplo. Primeiramente foi testada uma RdP simples, para que a geração do *XML* pudesse ser interpretada de maneira rápida. Os resultados podem ser vistos na figura 3.9.




```
<net id="rede">
  <place id="L0" x="178.0" y="78.0" desc="L0" tamanho="20" capacidade="1"/>
  <place id="L1" x="177.0" y="191.0" desc="L1" tamanho="20" capacidade="1"/>
  <transition id="T0" x="228.0" y="133.0" tamanho="20" type="normal" desc="T0"/>
  <transition id="T1" x="124.0" y="131.0" tamanho="20" type="normal" desc="T1"/>
  <arc id="a0" sentido="0" source="L0" target="T0" type="standard"/>
  <arc id="a0" sentido="1" source="T0" target="L1" type="standard"/>
  <arc id="a0" sentido="0" source="L1" target="T1" type="standard"/>
  <arc id="a0" sentido="1" source="T1" target="L0" type="standard"/>
</net>
```

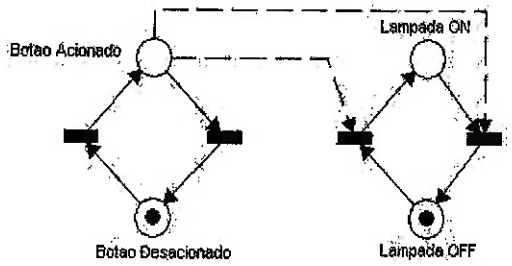
Figura 3.9: Descrição da RdP em *XML*.

Para um teste mais apurado, foi utilizada uma rede de maior complexidade. Esta rede modela novamente uma atividade num sistema de automação, onde um botão liga uma lâmpada caso desligada, e desliga caso ligada. A figura 3.10 apresenta esta rede.

No Anexo A encontra-se a descrição, em notação *XML*, da RdP da figura 2.6.

3.7.3. Geração de programa de CP a partir de um *SIPN*

Como exemplo de teste de uma conversão automática de um modelo em RdP do tipo *SIPN* para um programa de CP, o modelo da figura 2.6 foi editado com a ferramenta desenvolvida e, com base nesta rede foi gerado o programa *IL* para um controlador programável SIMATIC S7-300 fabricado pela empresa SIEMENS (S7-300). No Anexo B os resultados desta conversão podem ser analisados.




```
<net id="rede">
  <place id="Botao Acionado" x="136.0" y="86.0" desc="Botao Acionado" tamanho="20" capacidade="1"/>
  <place id="Botao Desacionado" x="135.0" y="200.0" desc="Botao Desacionado" tamanho="20" capacidade="1"/>
  <transition id="T0" x="186.0" y="142.0" tamanho="20" type="normal" desc="T0"/>
  <transition id="T1" x="82.0" y="140.0" tamanho="20" type="normal" desc="T1"/>
  <arc id="a0" sentido="0" source="Botao Acionado" target="T0" type="standard"/>
  <arc id="a0" sentido="1" source="T0" target="Botao Desacionado" type="standard"/>
  <arc id="a0" sentido="0" source="Botao Desacionado" target="T1" type="standard"/>
  <arc id="a0" sentido="1" source="T1" target="Botao Acionado" type="standard"/>
  <place id="Lampada ON" x="362.0" y="85.0" desc="Lampada ON" tamanho="20" capacidade="1"/>
  <place id="Lampada OFF" x="364.0" y="199.0" desc="Lampada OFF" tamanho="20" capacidade="1"/>
  <transition id="T2" x="284.0" y="144.0" tamanho="20" type="normal" desc="T2"/>
  <transition id="T3" x="445.0" y="141.0" tamanho="20" type="normal" desc="T3"/>
  <arc id="a0" sentido="1" source="T2" target="Lampada ON" type="standard"/>
  <arc id="a0" sentido="0" source="Lampada ON" target="T3" type="standard"/>
  <arc id="a0" sentido="1" source="T3" target="Lampada OFF" type="standard"/>
  <arc id="a0" sentido="0" source="Lampada OFF" target="T2" type="standard"/>
- <arc id="a0" sentido="0" source="Botao Acionado" target="T2" type="test">
  - <pontos>
      <ponto x="286.0" y="85.0"/>
    </pontos>
  </arc>
- <arc id="a0" sentido="0" source="Botao Acionado" target="T3" type="test">
  - <pontos>
      <ponto x="137.0" y="35.0"/>
      <ponto x="447.0" y="34.0"/>
    </pontos>
  </arc>
</net>
```

Figura 3.10: RdP de maior porte descrita em *XML*.

4. ESTUDOS DE CASO

Para fim de demonstração das funcionalidades da ferramenta desenvolvida, neste tópico foram avaliadas algumas aplicações da ferramenta desenvolvida.

4.1. Modelagem e Simulação - MiniCIM

O sistema CIM é um equipamento para fins de treinamento especializado da empresa Festo que, na configuração disponível no Laboratório de Sistemas de Automação do PMR-EPUSP é composto de 5 estações de trabalho, cada uma capaz de ser operada individualmente ou, no contexto de um sistema integrado e automatizado, cada estação possui certa autonomia na execução de suas tarefas. As estações de trabalho são as seguintes: Estação de Testes; Estação de Distribuição; Estação de Montagem com Unidade de Execução da Montagem; Sistema Inteligente de Transporte (SIT); Estação de Controle de Célula de Trabalho.

A Estação de Distribuição armazena as bases (cilindros) e as encaminha ao processo de produção de acordo com a demanda. Já a Estação de Testes é responsável pela identificação da cor (prateada, rosa ou preta) e teste da altura das bases (cilindros) vindas da estação de distribuição. Esta estação é também responsável por descartar peças rejeitadas neste teste. O Sistema Inteligente de Transporte conta com uma esteira de transporte e cinco carros (*pallets*). Este sistema de transporte tem pré-definido quatro locais distintos para a parada dos carros, sendo um deles destinado à Estação de Testes e um outro à Estação de Montagem. O SIT destina-se a transportar as bases (cilindros) identificadas e testadas da Estação de Testes para a Estação de Montagem, onde a montagem da peça é finalizada. As peças montadas são então transportadas pelo SIT para uma outra localidade distinta. A figura 4.1 representa o MiniCIM.

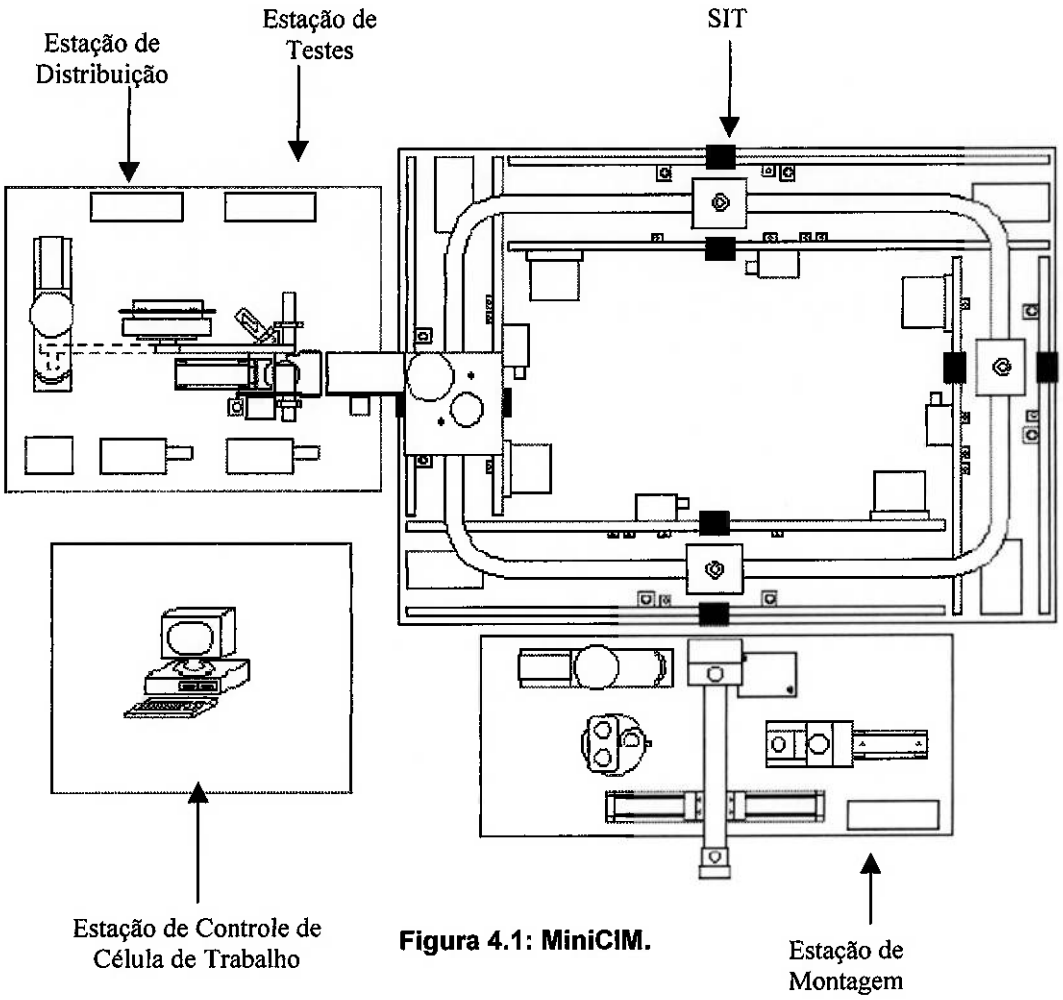


Figura 4.1: MiniCIM.

Este sistema foi modelado e simulado com a ferramenta desenvolvida e os resultados foram comparados com uma simulação realizada no *software* comercial disponível HPSim (Versão 1.1). A figura 4.2 mostra uma imagem de modelo pronto e a da simulação em andamento.

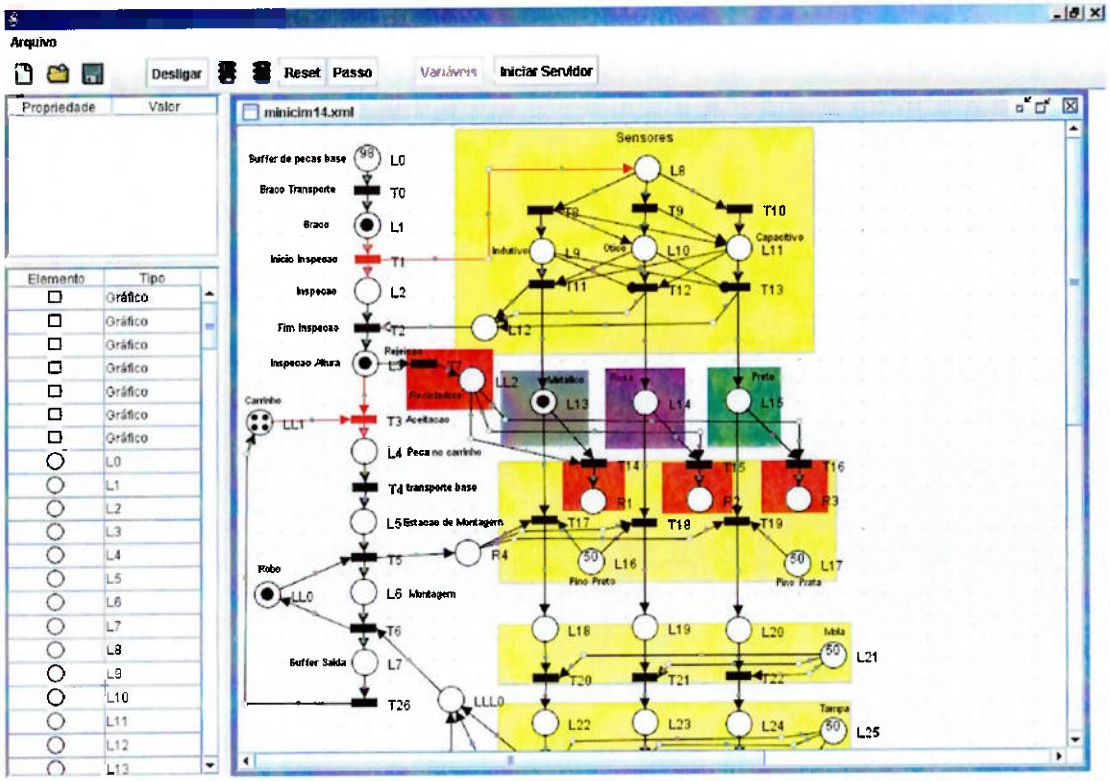


Figura 4.2: MiniCIM modelado na ferramenta desenvolvida.

Na figura a seguir, mostra-se a modelagem do MiniCIM no *software* comercial HPSim.

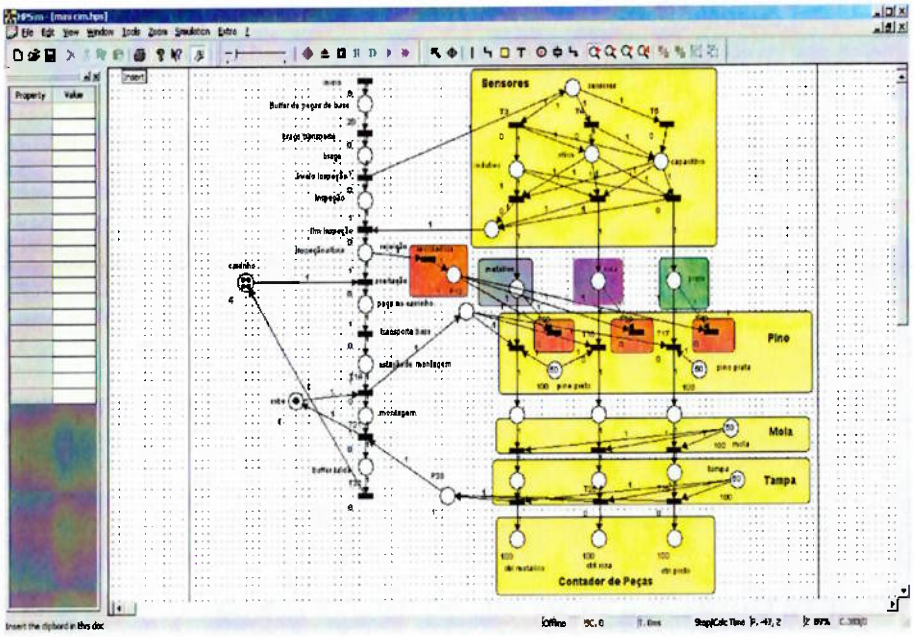


Figura 4.3: MiniCIM modelado no HPSim.

Após a modelagem, foi possível a simulação do sistema MiniCIM através do "jogador de marcas". Comparando-se o resultado com o esperado pelas regras da RdP, com o *software* comercial HPSim e com o algoritmo "jogador de marcas" encontrado ZHURAWASKIi & ZHOU (1994), garantiu-se que a funcionalidade de modelagem e simulação de RdP fosse plenamente alcançada.

4.2. Programação da Estação de Montagem do MiniCIM

A Estação de Montagem do MiniCIM é controlada através de um CP (SIMATIC S7-300, fabricado pela empresa SIEMENS). Este CP controla um robô manipulador que realiza a montagem das peças de acordo com as informações dos sensores da Estação de Distribuição.

As atividades do robô manipulador foram modeladas em RdP e através da ferramenta gerou-se o programa em *IL* para o CP desta estação. No Anexo B o código-fonte deste programa está disponível.

Para a geração deste programa, o tipo de RdP utilizado na modelagem foi *SIPN*. Com ela pode-se realizar o controle da movimentação do braço utilizando-se os sinais de entrada provenientes dos sensores e, baseando-se nestes, os sinais de saída correspondentes são gerados. A figura 4.4 apresenta a RdP que foi criada para controlar o braço, e que possibilitou a geração do programa em *IL* que foi carregado e executado no CP S7-300 da Estação de Montagem.

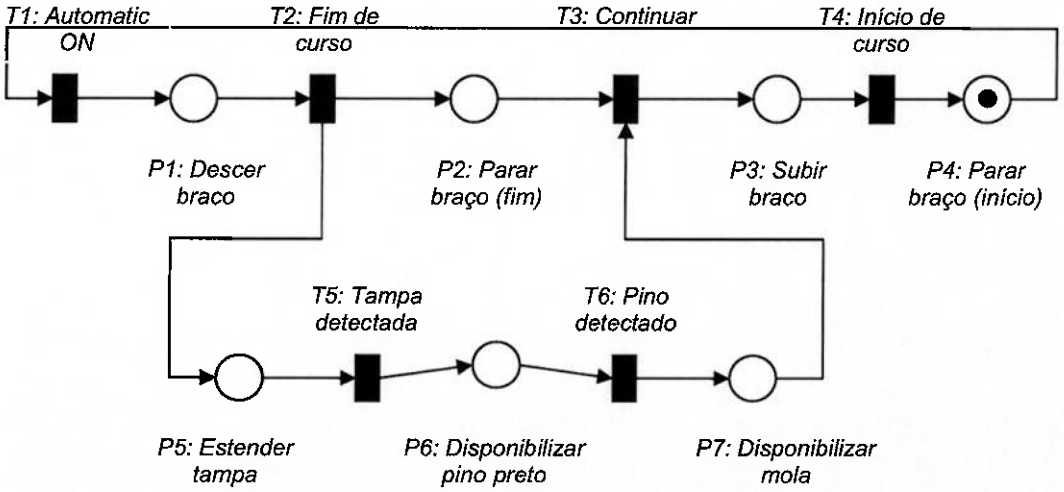


Figura 4.4: RdP que controla a movimentação do manipulador.

Nota-se que, além dos lugares P1, P2, P3 e P4 que representam os sinais de saída enviados ao CP, foram adicionados os lugares P5, P6 e P7 que indicam os sinais de saída enviados ao CP para que os componentes da peça produzida na Estação de Montagem sejam disponibilizados.

Com a RdP da figura 4.4 modelada na ferramenta e, após uma simulação executando-se o "jogador de marcas", verificou-se que dinâmica de movimentação do manipulador, bem como o processo de disponibilização dos componentes de montagem da peça haviam sido modelados corretamente.

Foi, então, gerado o programa de controle em *IL* que foi diretamente carregado e executado no CP S7-300 da Estação de Montagem do MiniCIM. A figura 4.5 apresenta a interface da ferramenta com a RdP da figura 4.4.

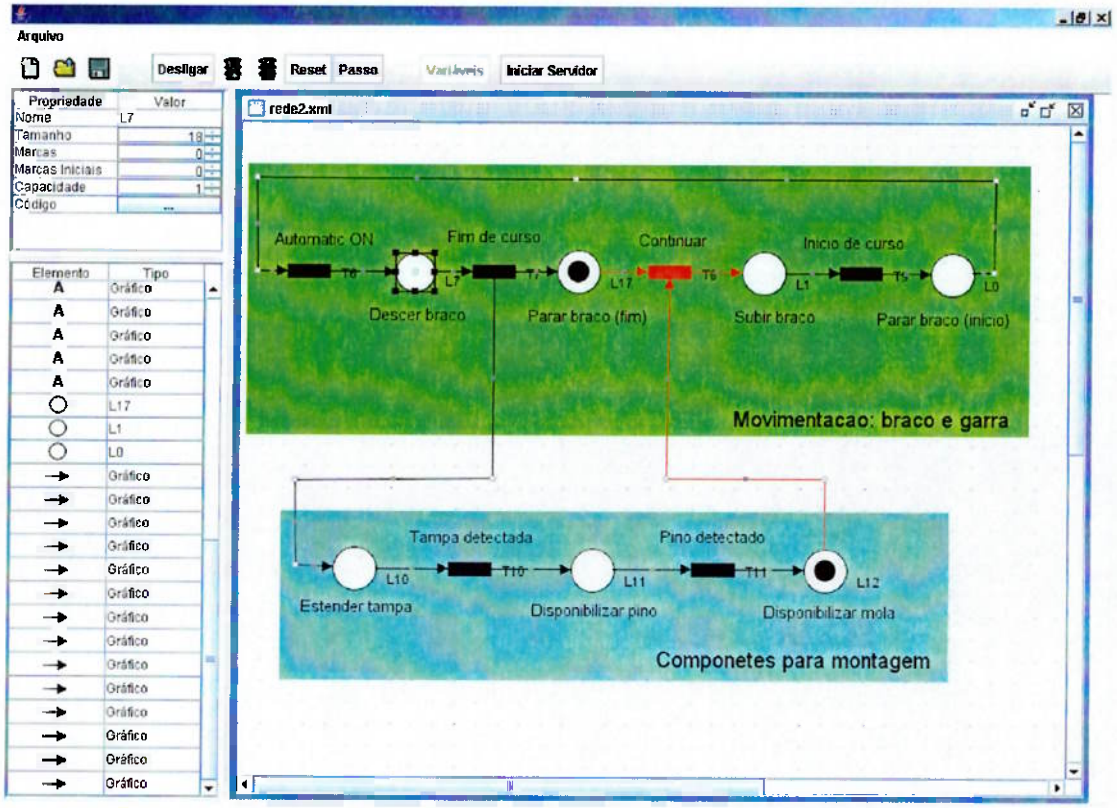


Figura 4.5: Simulação do modelo em RdP para controle da Estação de Montagem.

O código do programa de controle gerado neste estudo de caso consta no Anexo C.

4.3. Supervisão remota

Para realização de uma supervisão / monitoração remota utilizando-se a ferramenta desenvolvida, foi utilizado um sistema comumente encontrado em instalações industriais (FREY & MINAS, 2001): acumulador de ar comprimido conectado a compressores e equipado com sensores de pressão (vide figura 2.6). Na figura 4.6 a seguir, apresenta-se o modelo em RdP de um controlador para o acumulador.

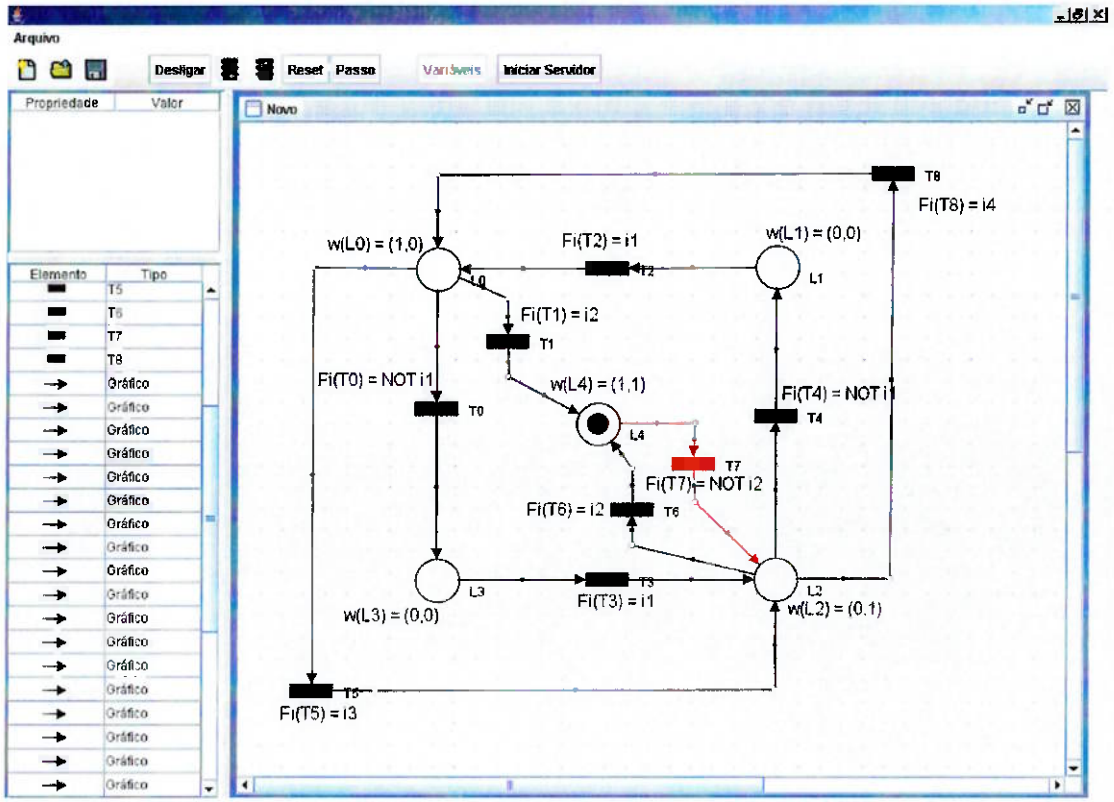


Figura 4.6: Simulação do modelo em *SIPN* do acumulador.

Através de comandos e janelas específicas da ferramenta desenvolvida, é possível realizar a supervisão de um sistema de controle modelado em RdP. Para isso, variáveis declaradas na ferramenta como sinais de entrada e de saída do sistema modelado são ilustradas na forma de botões (caso sejam sinais de entrada) e imagens de *leds* (caso sejam sinais de saída).

Para o uso desta característica em uma aplicação prática, a estratégia de controle modelada na RdP do tipo *SIPN* apresentada na figura 4.6 foi construída no *software* desenvolvido. Os sinais de entrada (i1, i2, i3 e o4) e saída (o1 e o2) puderam ser visualizados em um computador remoto, através de conexão via *Internet*.

O usuário remoto identificou a conexão através de endereços de rede *IP* (*Internet Protocol*), permitindo ao *software* a recuperação e atualização dos sinais do modelo. Foi possível também a sua interação com o modelo, alterando as variáveis de sinais de entrada.

Abaixo seguem imagens da ferramenta em funcionamento no computador com o modelo (figura 4.7), e no computador com acesso remoto (figura 4.8).

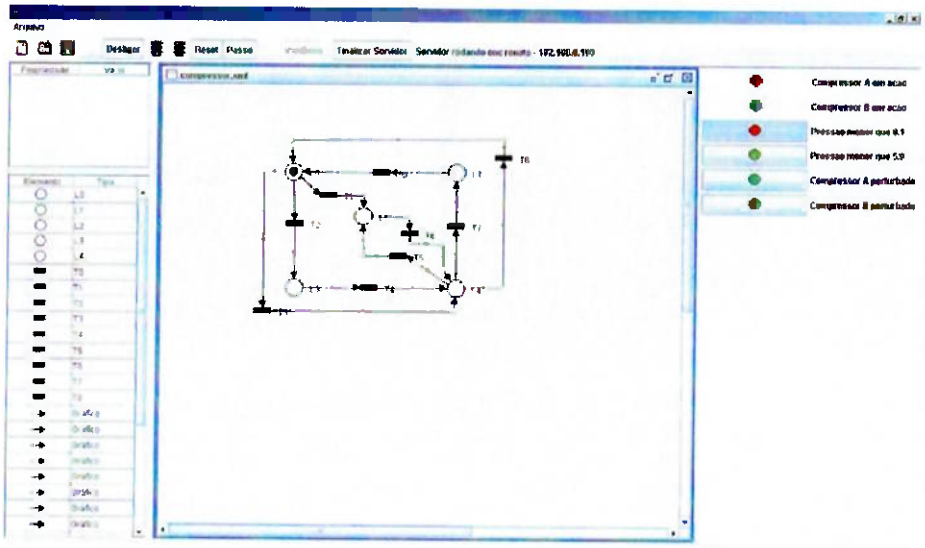


Figura 4.7: Software com o sistema modelado.

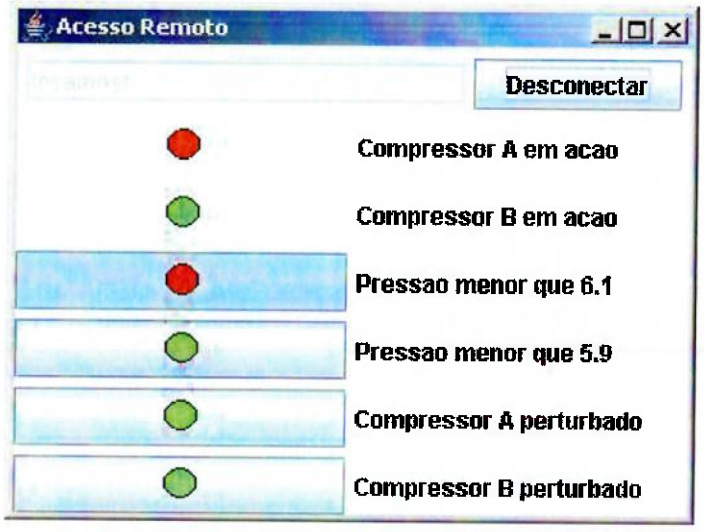


Figura 4.8: Software com acesso remoto.

5. CONCLUSÃO

Neste trabalho foi desenvolvida uma ferramenta que permite criação e edição de modelos em rede de Petri (RdP) de sistemas de automação. Os tipos de RdP que a ferramenta disponibiliza para a modelagem são as redes Condição-Evento (CE), Lugar-Transição (LT), e as redes de Sinal Interpretado (*SIPN*).

Dentre as funcionalidades da ferramenta desenvolvida, quatro são consideradas principais. A primeira delas é a possibilidade de edição gráfica dos modelos criados. Para fins de verificação do comportamento destes modelos, a ferramenta também permite que eles sejam executados de acordo com as regras da RdP. Nesta execução, pode-se observar a dinâmica do sistema, representado pelo modelo, através da alteração das marcações na RdP. Assim, é possível estudar o comportamento do sistema modelado. Caso o modelo não apresente o comportamento desejado, pode-se reeditá-lo e executá-lo novamente.

A segunda funcionalidade que a ferramenta desenvolvida neste projeto possui é a capacidade de programação de controladores. Com a ferramenta pode-se desenvolver um programa de controle de um controlador programável (CP) graficamente, utilizando *SIPNs*. Aos lugares do modelo em RdP são atribuídas instruções de comando que geram sinais de saída processados pelo CP e enviados ao objeto de controle e/ou operador envolvido, através de dispositivos de atuação e dispositivos de monitoração, respectivamente. Em resposta, o objeto de controle envia sinais através dos dispositivos de detecção para o modelo, isto é, como condições para o disparo das transições. Estes sinais são processados e estabelecem a mudança de estado, ou seja, a alteração da marcação na RdP. O programa de controle desenvolvido com a ferramenta pode ser convertido para uma linguagem adequada ao uso em CPs, e diretamente carregado em um CP para execução da tarefa de controle. A linguagem de CP adotada é o *Instruction List*, que é um dos padrões internacionais para o CP.

A terceira funcionalidade é o armazenamento dos modelos criados em formato *XML*. Após editar e salvar o modelo, a ferramenta gera um arquivo com a extensão ".xml" o qual contém a descrição da RdP desenvolvida. A razão para o uso deste formato de armazenamento é o intercâmbio de modelos de SEDs entre ferramentas / *softwares* que trabalhem com RdPs através de um formato pradonizado e aceito internacionalmente.

A quarta funcionalidade principal que destaca-se na ferramenta implementada é módulo de supervisão remota. Com este módulo tem-se uma interface gráfica, em RdP, para monitorar um sistema geograficamente distante, via *Internet*. Os sinais de saída de um CP são processados e enviados através da Internet para um computador com a ferramenta desenvolvida e esta, através de uma interface gráfica apropriada, apresenta a RdP que descreve o comportamento do respectivo sistema.

Realizando-se uma análise comparativa de funcionalidades da ferramenta implementada neste trabalho e de outros *softwares* especializados na modelagem de SEDs baseando-se na técnica de RdP, foi possível gerar uma tabela que relaciona as características destes *softwares* e da ferramenta. A tabela 5.1 traz esta relação.

Tabela 5.1: Relação de funcionalidades dos softwares.

FUNCIONALIDADE	*SOFTWARE*				
	FERRAMENTA DESENVOLVIDA	HPSIM	SIPN EDITOR	PIPE2_V2	PNK2.2
QUANTO À MODELAGEM					
EDITAR	SIM	SIM	SIM	SIM	SIM
SALVAR	SIM	SIM	SIM	SIM	SIM
RECUPERAR	SIM	SIM	SIM	SIM	SIM
QUANTO À SIMULAÇÃO					
EXECUTAR ("JOGADOR DE MARCAS")	SIM	SIM	NÃO	SIM	SIM
PAUSAR	SIM	SIM	NÃO	SIM	SIM
CONTINUAR	SIM	SIM	NÃO	SIM	SIM
QUANTO À SUPERVISÃO					
REMOTA	SIM	NÃO	NÃO	NÃO	NÃO
LOCAL	SIM	SIM	SIM	SIM	SIM
GERAÇÃO DE CÓDIGO PARA CPS					
INSTRUCTION LIST	SIM	NÃO	SIM	NÃO	NÃO

STRUCTURED TEXT	NÃO	NÃO	NÃO	NÃO	NÃO
LADDER	NÃO	NÃO	NÃO	NÃO	NÃO
SFC	NÃO	NÃO	NÃO	NÃO	NÃO
FORMATO PARA INTERCÂMBIO DE MODELOS					
XML	SIM	NÃO	SIM	NÃO	SIM

Observando-se esta tabela, nota-se que a ferramenta desenvolvida disponibiliza, em princípio, todas as funcionalidades que os demais *softwares* apresentam. No entanto, para aplicações que necessitem de monitoração remota e / ou geração de programas de controle em *IL* para CPs apenas a ferramenta apresentada neste trabalho atende a este requisito, dentre os *softwares* relacionados.

6. TRABALHOS FUTUROS

Como propostas para trabalhos futuros, têm-se os tópicos a seguir:

- Novas classes de RdP: dada a implementação estruturada da ferramenta, baseada nos conceitos de `classe`, `objeto`, `herança` e `polimorfismo` provenientes da orientação a objetos, pode-se reaproveitar o código-fonte já escrito com o intuito de ampliar o número de classes de RdP disponíveis para modelagem e simulação de SEDs de maior complexidade. Como exemplo, as classes *RTPN* e *CPN* poderiam ser incluídas como opções para representação de SEDs;

- Geração de programas de controle: no projeto desenvolvido, pode-se gerar um programa em *IL* para ser carregado e executado diretamente em um CP. No caso deste trabalho, o CP para o qual programas são gerados é o SIMATIC S7-300, fabricado pela Siemens. Contudo, a sintaxe da linguagem *IL* pode apresentar variações dentre os CPs comerciais disponíveis. Assim, pode-se adicionar à ferramenta módulos específicos de conversão de estratégias de controle para outras linguagens de programação, de acordo com o CP utilizado.

- Comunicação direta com CPs: outra possibilidade para um trabalho futuro seria o acréscimo à ferramenta de um módulo que ofereça uma interface de comunicação direta com um CP, isto é, sem o intermédio de um *software* comercial que realize esta comunicação. Este funcionalidade adicional permitiria que programas fossem diretamente carregados (*downloading*) e executados em um CP, ou lidos (*uploading*) dele diretamente da ferramenta.

7. REFERÊNCIAS BIBLIOGRÁFICAS

AGUILERA FERNANDES, EDSON. **Um modelo de referência para desenvolvimento de interfaces homem-computador,** dissertação de mestrado, Escola Politécnica da Universidade de São Paulo, 1992.

ALVES, G. R., COSTA, J. D., ARMELLINI, F. ET AL. **Petri Net´s Execution Algorithm for Applications in Manufacturing Systems Control**. Artigo Técnico, Universidade de São Paulo, 2003.

AMORY, A. PETRINI, J. J. **Sistema Integrado e Multiplataforma para Controle Remoto de Residências**. Trabalho de Conclusão de Curso, PUC-RS – Faculdade de Informática, Porto Alegre, RS, 2001.

ANGEL-RESTREPO, P. L. **Modelagem orientada a objetos de sistemas a eventos discretos: estudo de caso na síntese de controle de sistemas prediais**, dissertação de mestrado, Escola Politécnica da Universidade de São Paulo, 2004.

ARAKAKI, J., **Análise de Sistemas de Manufatura através da Metodologia PFS/MFG e Regras de Produção**, dissertação de mestrado, Escola Politécnica da Universidade de São Paulo, 1993.

AURESIDE. **Associação Brasileira De Automação Residencial**. <http://www.aureside.org.br>. Acesso em: 14 abril 2006.

BASS, L. **User Interface Software.** Ed. Wiley, 1993.

BILLINGTON, J. ET AL. **The Petri Net Markup Language: Concepts, Technology and Tools**. Springer-Verlag, Berlin, Heidelberg, 2003.

BOLZANI, C. **Residências Inteligentes**. Editora Livraria da Física, 2004.

COCKBURN, ALISTAIR. **Writting Effective Use Cases.** Addison-Wesley Longman, 1999.

CULWIN, FINTAN. **Java in the C.S. Curriculum.** Workshop at the 27th ACM SIGCSE Conference. 1997.

Disponível em http://www.scism.sbu.ac.uk/jfl/sanjose/sanjose.html. Acessado em 20/06/06.

DAVIS, A., **201 Principles of Software Development**, McGraw-Hill, 1995.

DICESARE, F. HARHALAKIS, G., PROTH, J.M, SILVA, M., VERNADAT, F.B. **Practice of Petri Nets in Manufacturing**, Chapman & Hall, 1993.

DYKES, L. TITTEL, E. **XML for Dummies**. Wiley Publishing, Hoboken, NJ, 4th, 2005.

ERICKSON, THOMAS D. **The art of human-computer interface design.** Edited by Laurel, Brenda. Introduction to 'Creativity and Design' session. Editora Addison-Wesley Publishing Company, 1990.

FREY, G., MINAS, M. **Internet-based development of logic controllers using Signal Interpreted Petri Nets and IEC 61131**. In Proceedings 5th World Multi-Conference on Systematics and Informatics (SCI 2001), Vol. 3, Orlando (FL), USA, pp. 297-302. [Online]. Available: citeseer.nj.nec.com/451425.html

HORTON, I. **Beginning Java 2: A comprehensive tutorial to Java programming**. Wrox Press, 2003.

JANDL JUNIOR, PETER. **Introdução ao Java.** Editora Berkeley, 2002.

JANDL JUNIOR, PETER. **Mais Java.** Editora Futura, 2003.

KAN, Stephen H., **Metrics and Models in Software Quality Engineering**, Second Edition. Ed. Addison Wesley Professional, 2002.

KERNIGHAN, B.W.; RITCHIE, D.M. *C,* **A Linguagem de Programação: Padrão ANSI**. Campus Ltda, Rio de Janeiro, 1990.

MERTKE, T., FREY, G. **Formal Verification of PLC-Programs Generated from Signal Interpreted Petri Nets.** In Proceedings of the IEEE Systems, Man, and Cybernetics Conference, 2001 [Online].

MIYAGI, P. E. **Controle Programável – Fundamentos do Controle de Sistemas a Eventos Discretos.** Editora Edgard Blucher, São Paulo, 1996.

MUNDOOO. **Mundo Orientação a Objeto.** <http://www.mundooo.com.br>. Acesso em: 02 maio 2006.

MURATA, T. **Petri nets: Properties, Analysis and Applications**. In Proceedings of the IEEE, Vol. 77, Nº 4, April de 1989 [Online].

REALI, A. H. C. **Redes de Petri - Lógica Computacional.** PCS0527, São Paulo, 2001.

MYERS, G., **The Art of Software Testing**, Wiley, 1979.

NETBEANS. **Site oficial Netbeans.** <http://www.netbeans.org>. Acesso em 05 maio 2006.

PRESSMAN, R. S. **Engenharia de Software**. Editora McGraw-Hill, 2002.

SUN MICROSYSTEMS. **Site oficial Java.** <http://java.sun.com>. Acesso em 05 maio 2006.

SOUZA, L. E. **Controladores Lógicos Programáveis**. FUPAI, 2001.

VILLANI, E. **Modelagem e Análise de Sistemas Supervisórios Híbridos.** Tese de Doutorado, EPUSP, São Paulo, SP, 2004.

ZHOU, M.; VENKATESH, K. **Modeling, simulation, and control of flexible manufacturing systems: a Petri net approach**. World Scientific Publishing Co. Pte. Ldta, Singapura, 1998.

ZHURUWASKI, R. AND ZHOU, M. **Petri nets and industrial applications: a tutorial.** IEEE Transactions on Industrial Electronics, v.41, n.6, p.567-583, December, 1994.

ANEXO A – Estratégia de controle descrita em notação XML

Exemplo de uma estratégia de controle, descrita em notação XML, modelada por uma RdP do tipo *SIPN*. A RdP utilizada é apresentada na figura 2.6.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
        <net id="n2">
                <name desc="acumulador de ar comprimido"/>
                <signals>
                        <var type="input" id="i1">pressure_lower_than_6.1_bar</var>
                        <var type="input" id="i2">pressure_lower_than_5.9_bar</var>
                        <var type="input" id="i3">compressor_A_disturbed</var>
                        <var type="input" id="i4">compressor_B_disturbed</var>
                        <var type="output" id="o1">compressor_A_runnning</var>
                        <var type="output" id="o2">compressor_B_runnning</var>
                </signals>
                <place id="p1" x="50" y="150" desc="p1" offx="0" offy="22">
                        <initialMarking marking="false" offx="-20" offy="-10"/>
                        <output desc="p1:o1=1;o2=0" offx="0" offy="-24"/>
                        <code>
                                <statement>
                                        <operator>S</operator>
                                        <signal>compressor_A_runnning</signal>
                                </statement>
                                <statement>
                                        <operator>R</operator>
                                        <signal>compressor_B_runnning</signal>
                                </statement>
                        </code>
                </place>
                <place id="p2" x="250" y="150" desc="p2" offx="-1" offy="20">
                        <initialMarking marking="false" offx="25" offy="5"/>
                        <output desc="p2:o1=0;o2=0" offx="-1" offy="-22"/>
                        <code>
                                <statement>
                                        <operator>R</operator>
                                        <signal>compressor_A_runnning</signal>
                                </statement>
                                <statement>
                                        <operator>R</operator>
                                        <signal>compressor_B_runnning</signal>
                                </statement>
                        </code>
                </place>
                <place id="p3" x="250" y="150" desc="p2" offx="-1" offy="20">
                        <initialMarking marking="false" offx="25" offy="5"/>
                        <output desc="p3:o1=0;o2=1" offx="-1" offy="-22"/>
                        <code>
                                <statement>
                                        <operator>R</operator>
                                        <signal>compressor_A_runnning</signal>
                                </statement>
                                <statement>
                                        <operator>S</operator>
```

```
                        <signal>compressor_B_runnning</signal>
                    </statement>
                </code>
            </place>
            <place id="p4" x="250" y="150" desc="p2" offx="-1" offy="20">
                <initialMarking marking="true" offx="25" offy="5"/>
                <output desc="p4:o1=0;o2=0" offx="-1" offy="-22"/>
                <code>
                    <statement>
                        <operator>R</operator>
                        <signal>compressor_A_runnning</signal>
                    </statement>
                    <statement>
                        <operator>R</operator>
                        <signal>compressor_B_runnning</signal>
                    </statement>
                </code>
            </place>
            <place id="p5" x="250" y="150" desc="p2" offx="-1" offy="20">
                <initialMarking marking="false" offx="25" offy="5"/>
                <output desc="p5:o1=1;o2=1" offx="-1" offy="-22"/>
                <code>
                    <statement>
                        <operator>S</operator>
                        <signal>compressor_A_runnning</signal>
                    </statement>
                    <statement>
                        <operator>S</operator>
                        <signal>compressor_B_runnning</signal>
                    </statement>
                </code>
            </place>

            <transition id="t1" x="150" y="50" type="normal" desc="t1" offx="0" offy="26">
                <delay interval="0" offx="7" offy="7"/>
                <input desc="i1" offx="-1" offy="-22"/>
                <code>
                    <statement>
                        <operator>AND</operator>
                        <signal>pressure_lower_that_6.1_bar</signal>
                    </statement>
                </code>
            </transition>
        <transition id="t2" x="150" y="250" type="normal" desc="t2" offx="-1" offy="22">
                <delay interval="0" offx="7" offy="7"/>
                <input desc="i2" offx="-1" offy="-22"/>
                <code>
                    <statement>
                        <operator>AND</operator>
                        <signal>pressure_lower_that_5.9_bar</signal>
                    </statement>
                </code>
            </transition>
            <transition id="t3" x="150" y="250" type="normal" desc="t3" offx="-1" offy="22">
                <delay interval="0" offx="7" offy="7"/>
                <input desc="i4" offx="-1" offy="-22"/>
                <code>
                    <statement>
                        <operator>AND</operator>
                        <signal>compressor_B_disturbed</signal>
```

```
                </statement>
        </code>
</transition>
<transition id="t4" x="150" y="250" type="normal" desc="t4" offx="-1" offy="22">
        <delay interval="0" offx="7" offy="7"/>
        <input desc="NOT i1" offx="-1" offy="-22"/>
        <code>
                <statement>
                        <operator>ANDN</operator>
                        <signal>pressure_low_that_6.1_bar</signal>
                </statement>
        </code>
</transition>
<transition id="t5" x="150" y="250" type="normal" desc="t5" offx="-1" offy="22">
        <delay interval="0" offx="7" offy="7"/>
        <input desc="i3" offx="-1" offy="-22"/>
        <code>
                <statement>
                        <operator>AND</operator>
                        <signal>compressor_A_disturbed</signal>
                </statement>
        </code>
</transition>
<transition id="t6" x="150" y="250" type="normal" desc="t6" offx="-1" offy="22">
        <delay interval="0" offx="7" offy="7"/>
        <input desc="NOT i2" offx="-1" offy="-22"/>
        <code>
                <statement>
                        <operator>ANDN</operator>
                        <signal>pressure_low_that_5.9_bar</signal>
                </statement>
        </code>
</transition>
<transition id="t7" x="150" y="250" type="normal" desc="t7" offx="-1" offy="22">
        <delay interval="0" offx="5" offy="4"/>
        <input desc="NOT i1" offx="-1" offy="-22"/>
        <code>
                <statement>
                        <operator>ANDN</operator>
                        <signal>pressure_low_that_6.1_bar</signal>
                </statement>
        </code>
</transition>
<transition id="t8" x="150" y="250" type="normal" desc="t8" offx="-1" offy="22">
        <delay interval="0" offx="5" offy="4"/>
        <input desc="i2" offx="-1" offy="-22"/>
        <code>
                <statement>
                        <operator>AND</operator>
                        <signal>pressure_low_that_5.9_bar</signal>
                </statement>
        </code>
</transition>
<transition id="t9" x="150" y="250" type="normal" desc="t9" offx="-1" offy="22">
        <delay interval="0" offx="4" offy="4"/>
        <input desc="i1" offx="-1" offy="-22"/>
        <code>
                <statement>
                        <operator>AND</operator>
                        <signal>pressure_low_that_6.1_bar</signal>
```

```
                    </statement>
                </code>
            </transition>

    <arc id="a1" x="75" y="75" source="t3" target="p1" type="standart"/>
        <arc id="a2" x="225" y="75" source="t1" target="p1" type="standart"/>
    <arc id="a3" x="225" y="225" source="p4" target="t1" type="standart"/>
    <arc id="a4" x="75" y="225" source="p1" target="t5" type="standart"/>
        <arc id="a5" x="75" y="75" source="p3" target="t3" type="standart"/>
        <arc id="a6" x="225" y="75" source="p1" target="t4" type="standart"/>
    <arc id="a7" x="225" y="225" source="p1" target="t2" type="standart"/>
    <arc id="a8" x="75" y="225" source="t2" target="p5" type="standart"/>
        <arc id="a9" x="75" y="75" source="t7" target="p4" type="standart"/>
        <arc id="a10" x="225" y="75" source="p5" target="t6" type="standart"/>
    <arc id="a11" x="225" y="225" source="t4" target="p2" type="standart"/>
    <arc id="a12" x="75" y="225" source="t8" target="p5" type="standart"/>
        <arc id="a13" x="75" y="75" source="t6" target="p3" type="standart"/>
        <arc id="a14" x="225" y="75" source="p3" target="t7" type="standart"/>
    <arc id="a15" x="225" y="225" source="p2" target="t9" type="standart"/>
    <arc id="a16" x="75" y="225" source="p3" target="t8" type="standart"/>
        <arc id="a17" x="75" y="75" source="t9" target="p3" type="standart"/>
        <arc id="a18" x="225" y="75" source="t5" target="p3" type="standart"/>
</net>
```

ANEXO B – Programa de controle em *Instruction List* gerado pela ferramenta

Este programa foi gerado a partir da RdP da figura 2.6. A sintaxe desta implementação é própria para a execução em controladores da família S7-300 fabricados pela empresa Siemens.

```
ORGANIZATION_BLOCK "Cycle Execution"
TITLE = "Main Program Sweep (Cycle) - Air Chamber Controller"
//CONTROLADOR PARA UMA "CÂMARA DE AR".
VERSION : 0.1

VAR_TEMP
  OB1_EV_CLASS : BYTE ; //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;   //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ; //Priority of OB Execution
  OB1_OB_NUMBR : BYTE ; //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;   //Reserved for system
  OB1_RESERVED_2 : BYTE ;   //Reserved for system
  OB1_PREV_CYCLE : INT ;    //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ; //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ; //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;   //Date and time OB1 started
  p1 : BOOL ;
  p2 : BOOL ;
  p3 : BOOL ;
  p4 : BOOL ;
  p5 : BOOL ;
  stab : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =n1
//Reinicializa a rede de Petri.
    A    "Initial_Marking";
    R    #p1;
    R    #p2;
    R    #p3;
    S    #p4;
    R    #p5;
    R    "compressor_A_running";
    R    "compressor_B_running";
NETWORK
TITLE =n2
//Caso alguma transição tenha disparado, a variável de estabilidade deve ser
//reinicializada para informar sobre a ocorrência do disparo.
L001: R    #stab;
NETWORK
TITLE =n3
//Verifica se a transição t1 é disparável. Se for, realiza-se o disparo e
//atualizam-se os pré e pós lugaes da rede.
// transition t1
    A    #p4;
    AN   #p1;
    A    "pressure_lower_6.1_bar";
```

```
      R    #p4;
      S    #p1;
      S    #stab;
 NETWORK
 TITLE =n4
 //Verifica se a transição t2 é disparável. Se for, realiza-se o disparo e
 //atualizam-se os pré e pós lugaes da rede.
 // transition t2
      A    #p1;
      AN   #p5;
      A    "pressure_lower_5.9_bar";
      R    #p1;
      S    #p5;
      S    #stab;
 NETWORK
 TITLE =n5
 //Verifica se a transição t3 é disparável. Se for, realiza-se o disparo e
 //atualizam-se os pré e pós lugaes da rede.
 // transition t3
      A    #p3;
      AN   #p1;
      A    "compressor_B_disturbed";
      R    #p3;
      S    #p1;
      S    #stab;
 NETWORK
 TITLE =n6
 //Verifica se a transição t4 é disparável. Se for, realiza-se o disparo e
 //atualizam-se os pré e pós lugaes da rede.
 // transition t4
      A    #p1;
      AN   #p2;
      AN   "pressure_lower_6.1_bar";
      R    #p1;
      S    #p2;
      S    #stab;
 NETWORK
 TITLE =n7
 //Verifica se a transição t5 é disparável. Se for, realiza-se o disparo e
 //atualizam-se os pré e pós lugaes da rede.
 // transition t5
      A    #p1;
      AN   #p3;
      A    "compressor_A_disturbed";
      R    #p1;
      S    #p3;
      S    #stab;
 NETWORK
 TITLE =n8
 //Verifica se a transição t6 é disparável. Se for, realiza-se o disparo e
 //atualizam-se os pré e pós lugaes da rede.
 // transition t6
      A    #p5;
      AN   #p3;
      AN   "pressure_lower_5.9_bar";
      R    #p5;
      S    #p3;
      S    #stab;
 NETWORK
 TITLE =n9
```

```
//Verifica se a transição t7 é disparável. Se for, realiza-se o disparo e
//atualizam-se os pré e pós lugaes da rede.
// transition t7
    A    #p3;
    AN   #p4;
    AN   "pressure_lower_6.1_bar";
    R    #p3;
    S    #p4;
    S    #stab;
NETWORK
TITLE =n10
//Verifica se a transição t8 é disparável. Se for, realiza-se o disparo e
//atualizam-se os pré e pós lugaes da rede.
// transition t8
    A    #p3;
    AN   #p5;
    A    "pressure_lower_5.9_bar";
    R    #p3;
    S    #p5;
    S    #stab;
NETWORK
TITLE =n11
//Verifica se a transição t9 é disparável. Se for, realiza-se o disparo e
//atualizam-se os pré e pós lugaes da rede.
// transition t9
    A    #p2;
    AN   #p3;
    A    "pressure_lower_6.1_bar";
    R    #p2;
    S    #p3;
    S    #stab;
NETWORK
TITLE =n12
//Se alguma transição foi disparada, tem-se stab=1. Logo realiza-se o jump.
// Check stability variable
    A    #stab;
    JC   L001;
NETWORK
TITLE =n13
//Se o lugar p1 estiver marcado, envias-se o(s) sinal(s) de saída
//correspondeste(s) a ele.
// place p1
    A    #p1;
    S    "compressor_A_running";
    R    "compressor_B_running";
NETWORK
TITLE =n14
//Se o lugar p2 estiver marcado, envias-se o(s) sinal(s) de saída
//correspondeste(s) a ele.
// place p2
    A    #p2;
    R    "compressor_A_running";
    R    "compressor_B_running";
NETWORK
TITLE =n15
//Se o lugar p3 estiver marcado, envias-se o(s) sinal(s) de saída
//correspondeste(s) a ele.
// place p3
    A    #p3;
    R    "compressor_A_running";
```

```
    S    "compressor_B_running";
NETWORK
TITLE =n16
//Se o lugar p4 estiver marcado, envias-se o(s) sinal(s) de saída
//correspondeste(s) a ele.
// place p4
    A    #p4;
    R    "compressor_A_running";
    R    "compressor_B_running";
NETWORK
TITLE =n17
//Se o lugar p5 estiver marcado, envias-se o(s) sinal(s) de saída
//correspondeste(s) a ele.
// place p5
    A    #p5;
    S    "compressor_A_running";
    S    "compressor_B_running";
END_ORGANIZATION_BLOCK
```

ANEXO C – Código em *IL* gerado pela ferramenta

Código em *IL* gerado pela ferramenta para programação da Estação de Montagem do MiniCIM, referente ao estudo de caso apresentado no item 4.2.

```
ORGANIZATION_BLOCK "Cycle Execution"
TITLE = "Main Program Sweep (Cycle)"
VERSION :

VAR_TEMP
        OB1_EV_CLASS : BYTE ; //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
        OB1_SCAN_1 : BYTE ; //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
        OB1_PRIORITY : BYTE ; //Priority of OB Execution
        OB1_OB_NUMBR : BYTE ; //1 (Organization block 1, OB1)
        OB1_RESERVED_1 : BYTE ; //Reserved for system
        OB1_RESERVED_2 : BYTE ; //Reserved for system
        OB1_PREV_CYCLE : INT ; //Cycle time of previous OB1 scan (milliseconds)
        OB1_MIN_CYCLE : INT ; //Minimum cycle time of OB1 (milliseconds)
        OB1_MAX_CYCLE : INT ; //Maximum cycle time of OB1 (milliseconds)
        OB1_DATE_TIME : DATE_AND_TIME ;   //Date and time OB1 started
        // variáveis do programa gerado a partir da rede de Petri.
        L0:BOOL;
        L1:BOOL;
        L2:BOOL;
        L3:BOOL;
        L4:BOOL;
        L5:BOOL;
        L6:BOOL;
        stab:BOOL;
END_VAR

BEGIN

NETWORK
TITLE =
// (Re)inicializa rede de Petri
        A       I12.3;
        R       #L0;
        R       #L1;
        R       #L2;
        S       #L3;
        R       #L4;
        R       #L5;
        R       #L6;
NETWORK
TITLE =
L001:   R       #stab;

NETWORK
TITLE =
// Transition T0
        A       #L3;
        AN      #L0;
        A       I12.0;   //X
        R       #L3;
        S       #L0;
```

```
        S       #stab;

NETWORK
TITLE =
// Transition T1
        A       #L0;
        AN      #L1;
        AN      #L4;
        A       I16.6;   //X
        R       #L0;
        S       #L1;
        S       #L4;
        S       #stab;

NETWORK
TITLE =
// Transition T2
        A       #L1;
        A       #L6;
        AN      #L2;
        A       I12.2;   //X
        R       #L1;
        R       #L6;
        S       #L2;
        S       #stab;

NETWORK
TITLE =
// Transition T3
        A       #L2;
        AN      #L3;
        A       I16.4;   //X
        R       #L2;
        S       #L3;
        S       #stab;

NETWORK
TITLE =
// Transition T4
        A       #L4;
        AN      #L5;
        A       I20.6;   //X
        R       #L4;
        S       #L5;
        S       #stab;

NETWORK
TITLE =
// Transition T5
        A       #L5;
        AN      #L6;
        A       I20.7;   //X
        R       #L5;
        S       #L6;
        S       #stab;

NETWORK
TITLE =
// Check stability variable
        A       #stab;
```

```
        JC      L001;

NETWORK
TITLE =
// Place L0
        A       #L0;
        S       Q16.5; //X
        S       Q16.1; //X
        R       Q12.1; //X
        S       Q16.2; //X

NETWORK
TITLE =
// Place L1
        A       #L1;
        R       Q16.5; //X
        R       Q16.1; //X
        R       Q16.2; //X

NETWORK
TITLE =
// Place L2
        A       #L2;
        S       Q16.4; //X
        S       Q16.0; //X
        S       Q16.3; //X
        S       Q20.4; //X
        R       Q20.1; //X
        S       Q20.0; //X

NETWORK
TITLE =
// Place L3
        A       #L3;
        R       Q16.4; //X
        R       Q16.0; //X
        R       Q16.2; //X
        R       Q16.3; //X
        S       Q12.1; //X
        R       Q20.5; //X
        R       Q20.4; //X
        R       Q20.1; //X
        R       Q20.0; //X
        S       Q20.2; //X

NETWORK
TITLE =
// Place L4
        A       #L4;
        S       Q20.5; //X

NETWORK
TITLE =
// Place L5
        A       #L5;
        S       Q20.1; //X
        R       Q20.5; //X

NETWORK
TITLE =
```

```
// Place L6
      A     #L6;
      R     Q20.2;  //X

END_ORGANIZATION_BLOCK
```

RESUMO

Neste relatório, procurare-se expor todos os avanços obtidos na elaboração da documentação, casos de uso, pesquisas bibliográficas e implementação do software para geração de massas de teste. No capítulo de resumo teórico, colocam-se alguns conceitos pesquisados e que servem como base para o desenvolvimento do software. Há uma discussão sobre os tipos de teste mais comuns, como caixa-preta e caixa-branca. Também abordamos algumas linguagens próprias existentes, qualidade de software representada pelo conceito de co-standard e uma discussão um pouco mais genérica sobre o teste em si: seus objetivos, a forma como pode ser encarado, como pode ser conduzido e fatores de sucesso e fracasso para o teste. Evidentemente que toda essa discussão será cercada pelo tema principal de nosso trabalho, ou seja, a geração de massas de teste. Em resumo, propõe-se aqui, desenvolver um software capaz de gerar massas de teste adequadas para condução futura de homologações. Em geral, o teste de um software, em ambiente profissional ou não, se resume a geração de uma massa de teste genérica e verificação de algumas funcionalidades básicas, em especial, aquelas que interagem diretamente com o usuário. Tal metodologia resulta em produtos que atendem parcialmente os requisitos ou apresentam problemas durante sua vida útil. Além disso, há manutenção, retrabalho e descontentamento por parte do usuário. O software proposto fará a verificação de como o alvo do teste lida com as entradas e se as saídas são satisfatórias (teste caixa-preta). Em anexo, temos todos os casos de uso desenvolvidos, já com exemplos de telas,. Casos dentro do escopo deste trabalho são apresentados e discutidos.

Palavras-chave: Engenharia de Software. Testes. Massa de Testes. Java. Casos de Uso.

Abstract

At this report, we try to expose all of the advances concerning documentation, use cases, bibliographic research and program implementation. In the chapter of theorical briefing, there are some concepts investigated that were the base for the software development. We discuss about the two principal types of tests: Black box and White Box. We also discuss about software quality, represented by the concept of co-standard and a more generic approach on software test: Objectives, the way it can be faced, how can we conduct it and which are the factors of success and failure. To sum up, we seek for the creation of a software capable of generating a reliable and relevant mass of test used for homologation. Nowadays, mass of tests are created without professional parameters, just for the test of basic functionalities, especially those related with interface. This behavior results in products that just partially meet users requirements and always have problems during its life time. Furthermore there is always the need for rebuilding and the user getting angry. The appendix presents all the use cases and UML diagrams developed during our work. Cases within the scope are discussed and analyzed.

Key Words: Software Engineering. Tests. Mass of tests. Java. Use Cases

Tabela de Figuras

SUMÁRIO

1. Introdução

O presente trabalho tem por objetivo apresentar alternativas para testes de software, campo vasto e ainda pouco explorado devido a precariedade de estudos e materiais sobre o assunto.

Nosso foco se dará na geração de massa de testes. Atualmente, a geração da massa de testes é feita de modo pouco profissional, por não dizer, amador. Não há nenhum método na seleção dos registros que serão retirados da base e usados no teste. A não ser que haja um ambiente de homologação, os dados são escolhidos, normalmente, por serem os primeiros da tabela. Tal metodologia resulta em produtos que atendem parcialmente os requisitos ou apresentam problemas durante sua vida útil. Além disso, há manutenção, retrabalho e descontentamento por parte do usuário. Mas como garantir que tais dados são realmente relevantes? Como garantir que a carga de registros utilizada é suficiente? Por outro lado, como garantir que a massa de testes gerada não é excessiva e que esforço e dinheiro estão sendo desperdiçados? Enfim como garantir que a massa de testes fornecerá condições para um teste satisfatório?

Neste momento pode-se expandir a discussão. Uma primeira abordagem que deve ser compreendida por desenvolvedores de softwares é o por que do teste do software. A grande maioria dos desenvolvedores acredita que o objetivo do teste de software é provar que o mesmo funciona, mas isso é um grande erro. Ao se tentar verificar que uma simples conta de soma realmente está correta, precisamos verificar se todas as somas

possíveis estão corretas, afim de garantir cem por cento de certeza, o que é simplesmente inviável. Assim, conclui-se que o foco do teste está errado. O verdadeiro objetivo do teste de software não é provar o que funciona corretamente, mas sim encontrar erros, também chamados de Bugs. É neste ponto que uma massa de testes gerada com parcimônia ajuda no teste.

Alguns críticos propalam que o teste de software pode consumir um tempo excessivo do projeto, aumentando de forma desproporcional o custo do mesmo. Porém é comprovado que um teste profissional, orientado, conduzido durante todo o projeto e, principalmente, otimizado, reduz o retrabalho e manutenção diminuindo os custos. Um software de geração de massas de testes caminha lado a lado com este conceito, pois serve como ferramenta para a já citada otimização. A seguir detalhamento do trabalho.

2. Resumo Teórico

No curso de graduação em engenharia tem-se dado muito enfoque em toda a sorte de métodos e processos de cálculo que são essenciais para a realização de projetos de engenharia. Por conseguinte, o aluno tem um grande conhecimento sobre Cálculo Integral, Cálculo Variacional, teorias de controle clássicas e modernas, cálculo estatístico e tantas outras ferramentas úteis para analises numéricas sobre os problemas. Porém, essa estrutura curricular tem como objetivo final formar uma mente "engenheira" .

Apesar dessa estrutura formar profissionais de altíssimo nível, o assunto que discutiremos nesse tópico também é de grande utilidade para um engenheiro moderno, mesmo não possuindo a estrutura clássica supracitada.

Em uma primeira análise, parece que estamos diante de um paradigma de pouca utilidade. Porém, o uso de técnicas não tão focadas em cálculos e mais cálculos tem se mostrado uma ferramenta poderosa na análise de sistemas alto grau de complexidade se os aplicarmos de forma correta em vários âmbitos do mesmo.

A caixa preta consiste de uma máquina com uma ou mais entradas que processa uma ou mais saídas. Ela simplifica muito a análise de um sistema, já pensando em projetos de software, pois a função que ela substitui pode ter qualquer grau de complexidade, desde um simples filtro passa baixa em uma automação industrial, até equações de transferência de calor multidimensionais em um bloco de motor.

A área da engenharia em que essa forma de pensamento é mais utilizada é na eletrônica. Nos primeiros microprocessadores produzidos existiam pouco mais de 2000 transistores para serem polarizados. Por conta disso um engenheiro daquela sabia exatamente qual o numero do transistor deveria ser polarizado para que a lógica que ele necessitava fosse feita.

Com o passar dos anos o número de transistores encapsulados em um circuito cresceu exponencialmente e por conta disso não havia mais como um único profissional ter todos os transistores do circuito integrado em sua memória. Com isso teve-se que criar algumas caixas pretas para simplificar o processo, surgindo as portas lógicas E, OU, OU-EXCLUSIVO e outras para simplificar o trabalho.

Mas, com o tempo, outras caixas pretas tiveram que ser pensadas para abstrair mais o pensamento e com isso formaram-se os flip-flops, Unidade Lógicas Aritméticas(ULA) e assim por diante. Seguindo a mesma

linha de raciocínio, podemos dizer que o grau de abstração na programação chegou hoje às linguagens orientadas a objeto com, por exemplo, o Java, na qual uma simples instrução de soma é na verdade uma caixa preta que comanda diversos operadores lógicos que por sua vez comandam uma serie de transistores para realizar as operações.

Através desses exemplos podemos perceber o quanto já nos utilizamos dessa ferramenta sem perceber e vimos o poder de se olhar por através dela. Nas próximas páginas veremos como podemos utilizar esse conceito, entre outros, na geração de massas de teste.

2.1 Objetivo do teste

Um assunto muito pouco compreendido por desenvolvedores de softwares é o por que do teste do software. A grande maioria dos desenvolvedores acredita que o objetivo do teste do software é provar que o mesmo funciona, mas isso é um grande erro.

Se tentarmos verificar que uma simples conta de soma realmente está correta devemos verificar se todas as somas possíveis estão corretas, o que é simplesmente inviável. Por isso vemos que o foco do teste está errado. O verdadeiro objetivo do software não é provar que ele funciona corretamente, o verdadeiro sentido do teste é encontrar os erros do software, ou seja, os famigerados Bugs.

Essa afirmação nos leva a derrubar mais um conceito sedimentado na maioria dos desenvolvedores que é o de que "Um teste bem sucedido é aquele que não resulta em nenhum erro". A forma correta de se analisar um bom teste é pela quantidade de erros encontrados, e quanto mais erros

melhor! Assim temos que perceber que um teste que não resulte em nenhum erro foi um teste ruim, pois gastou tempo de desenvolvimento e não agregou valor ao projeto.

Devemos também sempre nos lembrar que as características básicas que norteiam os testes são:

- O software deve atender integralmente aos requisitos do cliente
- Um teste completo é impossível
- Deve-se testar primeiramente pequenas funções isoladas, para depois testarmos módulos completos.
- Sempre que possível os testes devem ser projetados e realizados por pessoas não envolvidas em sua programação.

2.2 Tipos de teste

Há uma grande gama de tipos de teste, todos buscando cobrir a maior quantidade de casos de software. Testes de Integração, Testes de Stress, Testes de Função... todos sendo utilizados de acordo com a necessidade do usuário. Porém, há duas estratégias de teste reconhecidamente universais e comumente utilizadas, independente do tipo de teste que se está conduzindo. São elas Teste Caixa-Preta e Teste Caixa-Branca. A seguir um pequeno resumo destas duas estratégias, e como se relacionam com nosso programa.

a) Teste Caixa-Preta: O teste caixa preta consiste em uma verificação de fronteiras do software. O testador fornece as entradas necessárias e espera pela saída, comparando com a esperada. Se há qualquer problema, o usuário pode verificar se a entrada está contaminada. Em caso negativo, o

problema é interno e outros testes deverão ser conduzidos. Nosso software permite ao testador gerar bases de dados para este tipo de teste, basta apenas cadastrar um cenário que contemple apenas as fronteiras.

b) Teste Caixa-Branca: O teste caixa branca é mais invasivo do que o caixa preta, exigindo do testador mais tempo, conhecimento do código e experiência em teste de software. Neste caso, todo o código do programa será varrido. Limites de funções serão testados, relacionamentos, acessibilidade... Em última análise, podemos encarar o teste caixa branca como uma sucessão de "n" testes caixa preta em menor escala. É neste tipo de teste que se encontram erros de lógica e sintaxe, já apontados no teste caixa preta. Para gerar massas de teste para este tipo de aplicação, o usuário deve cadastrar como cenário apenas a função ou conjunto de funções alvo, trabalhando da mesma forma que no caso anterior.

2.3 Particularidades da divisão da programação de softwares

Nos dias de hoje a maior parte dos projetos desenvolvidos são elaborados por mais de uma pessoa ao mesmo tempo, o que é de grande valia quando pensamos que todas as pessoas que estão colaborando unem seus esforços em prol de um objetivo comum. Essa afirmação vale para todas as áreas de atuação, tanto em uma mesa de cirurgia, onde diversos profissionais trabalham de diferentes formas ao mesmo tempo, quanto na elaboração de um projeto cientifico. Mas será que essa mesma afirmação é correta na elaboração de softwares?

É inútil tentar supor que tal afirmação não seja válida na construção de software, pois não há como imaginar como uma única pessoa poderia desenvolver sozinha um sistema com mais de algumas mil linhas de código, que dirá, então, um programa como um sistema operacional que normalmente apresenta algumas dezenas de milhões de linhas de código. Portanto, vamos tentar provar que é possível a união de diversas pessoas na programação de um único software, mostrando abaixo como fazer para que isso seja possível.

O principal problema que uma equipe de programação pode enfrentar durante um projeto é como garantir que toda a equipe escreverá um código que tenha sentido quando unificado e como garantir que a linguagem que uma pessoa está utilizando para escrever seu código segue o mesmo princípio da linguagem utilizada pelo resto da equipe. Para que isso seja possível é necessário que se estabeleça alguns parâmetros básicos antes da equipe sair escrevendo códigos isoladamente, como veremos abaixo:

A)Linguagem Padrão: Uma linguagem padrão especifica a sintaxe e a semântica de uma linguagem de programação (por exemplo Cobol, Ada, C++) ou em uma linguagem de sistemas (por exemplo SQL). Um padrão de linguagem é composto de um conjunto de regras de sintaxe que informam como devemos nomear as variáveis que serão utilizadas, qual será a estrutura de dados que será utilizada, quais mneumônicos serão utilizados dentre outras regras que farão com que cada programador saiba facilmente ler os códigos escritos por seus colegas e saiba como integrar esses códigos com o menor retrabalho possível.

B)Protocolo Padrão: O protocolo padrão especifica para os programadores qual deve ser a estrutura de envio e recebimento de mensagens que deve ser utilizado quando tivermos que fazer dois sistemas diferentes se comunicarem automaticamente, isso evita que cada programador crie seu próprio protocolo o que faria com que em um mesmo sistema diversos protocolos diferentes fossem utilizados aumentando a complexidade do software desnecessariamente e fazendo com que o mesmo programado tivesse que programar tanto o envio das mensagens como o recebimento das mesmas.

C) Bibliotecas Padrões: A estipulação de bibliotecas padrões é necessária para evitar que inconsistências de falta de biblioteca ocorram entre os programadores assim, se estipulam quais bibliotecas poderão ser utilizadas e qual versão das mesmas serão utilizadas. Dessa forma, qualquer recurso que um programador precise e não esteja nas bibliotecas deverá ser escrito como uma função do programa, fazendo com que qualquer programador da equipe possa compilar o programa sem erros dessa ordem.

3. O Software

O software se divide em, basicamente, dois módulos: Telas de cadastro e Núcleo de geração de massas de testes e criação do arquivo de saída. Porém, antes de explicitar o comportamento de cada módulo, cabe ressaltar a estratégia de imlementação utilizada.

3.1 Estratégias de Implementação

Os dois módulos foram construídos baseados no modelo de três camadas, ou seja, classes especializadas que se comunicam por um DataObject. O DataObject consiste em uma classe formada apenas por "getters" e "setters", responsável pelo transporte de dados entre as camadas e com o banco de dados.

As camadas se apresentam da seguinte forma:

A) Camada de Dados: Formada por classes responsáveis pela comunicação com banco de dados, se valendo dos dados do DataObject. Todas as classes que necessitam de informações do banco de dados devem acionar sua respectiva camada de dados.

B) Camada de Negócios: Camada intermediária, possui toda a inteligência do aplicativo, seja em termos de cadastro, seja representando o "core" do programa. Recebe os dados e requisições da camada de interface através do DataObject e os trabalha disparando requisições para a camada de dados, tanto de inserção quanto de consulta, deleção ou alteração.

C) Camada de interface: Essa camada faz toda a interação com o usuário, transmitindo as informações fornecidas por este. Estas informações podem ser dados, como nomes ou valores ou requisições como salvar ou deletar.

As vantagens deste modelo se encontram na garantia da consistência dos dados, que trafegam através de uma classe específica, da garantia de

qualidade do banco de dados que tem seu acesso controlado pela classe de dados e, principalmente, pelo desacoplamento de funções que permite que se façam manutenção ou upgrades em determinadas classes e funções sem necessidade de alteração de todo o código.

3.2 Módulo de Cadastro

O programa encadeia informações de cadastro com dois objetivos. O primeiro é montar uma imagem lógica do banco de dados que o usuário usará como fonte para a massa de testes. O segundo consiste na construção dos cenários de testes pretendidos pelo usuário, entendendo quais são os objetivos e, consequentemente, selecionando os registros mais adequados. A seqüência de cadastro é a seguinte:

1) Cadastro do projeto

   Usuário cadastra o projeto e o caminho para o banco de dados usado.

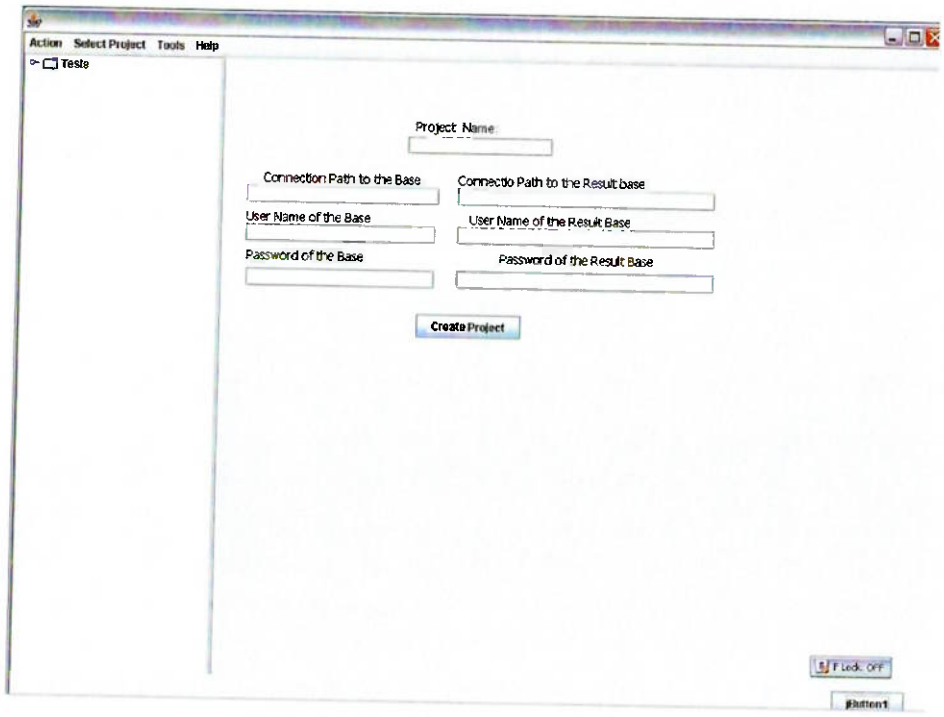


Figura 1 - Tela de cadastro de Projeto

2) Cadastro de Cenário

Usuário cadastra um cenário de testes, que será referência para o tipo de teste conduzido.

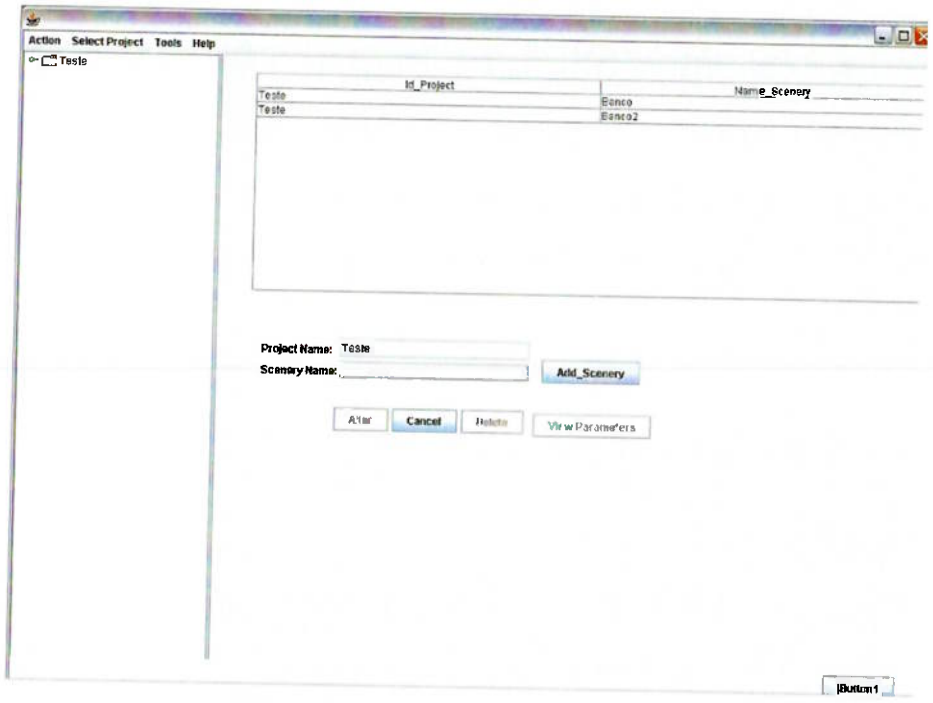


Figura 2 - Tela de cadastro de Cenários

3) Cadastro de Casos

Usuário cadastra os casos pertencentes ao cenário, associados a regra de consulta

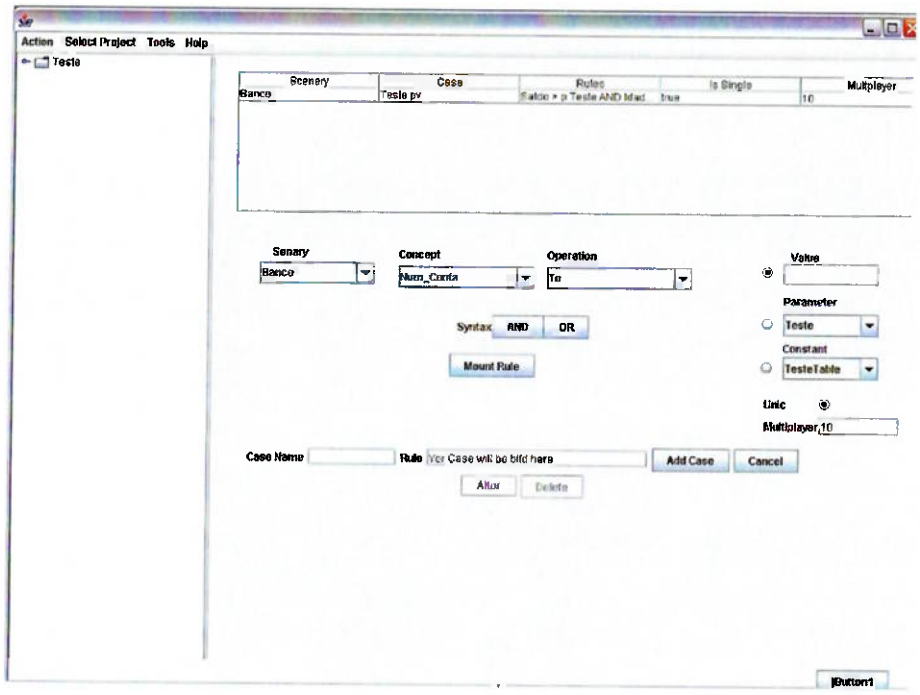


Figura 3 - Tela de cadastro de casos

4) Cadastro de regras

Usuário cadastra regras associadas aos casos, que os especificam e personalizam.

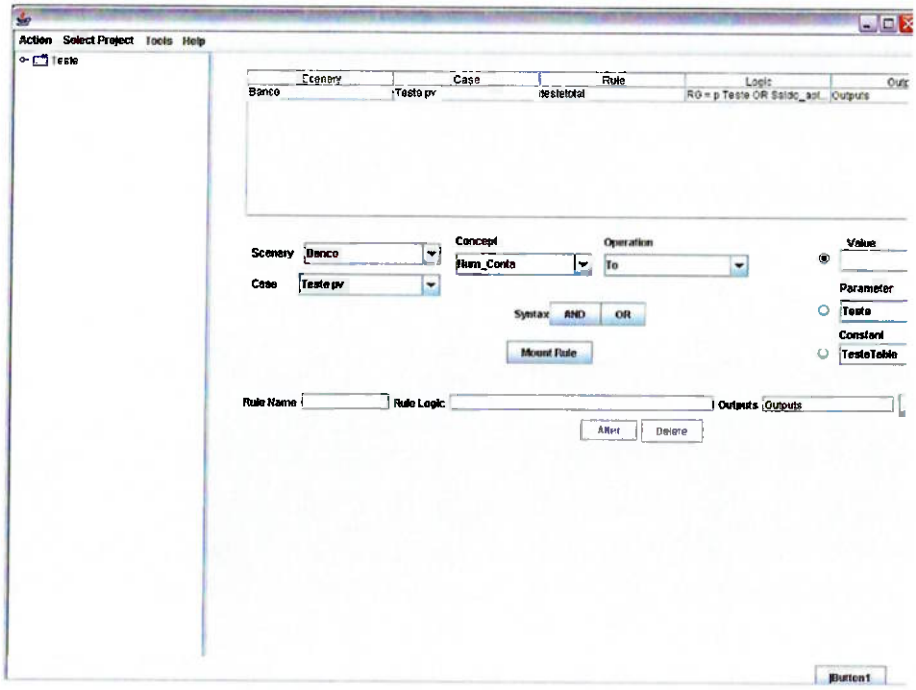


Figura 4 - Tela de cadastro de Regras

5) Cadastro de parâmetros

Usuário associa nomes a conjunto de valores que serão largamente utilizados, poupando trabalho de recadastro.

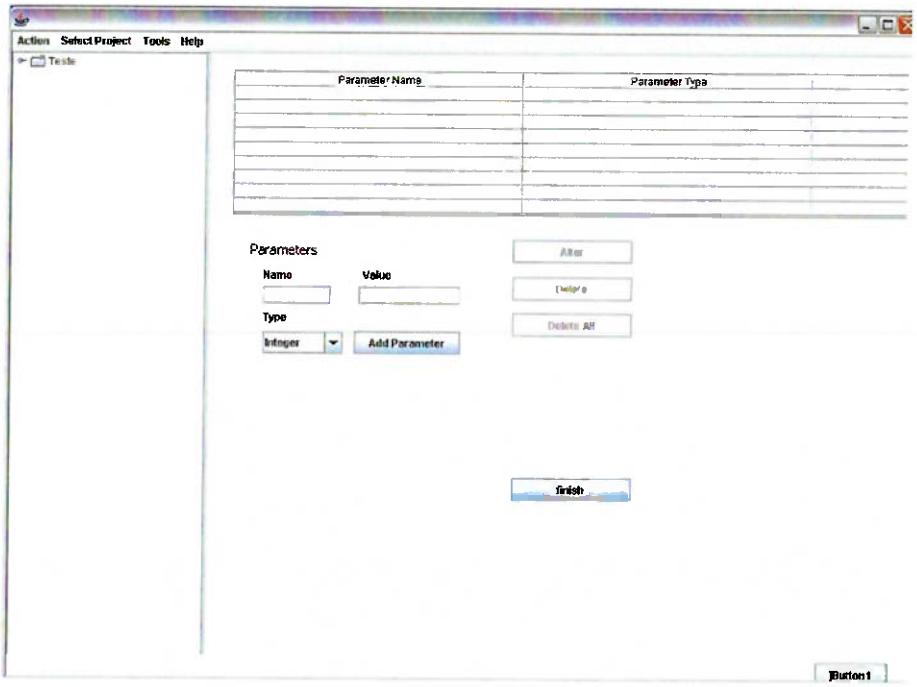


Figura 5 - Tela de cadastro de parâmetro

6) Cadastro de Constantes

Usuário atribui nome a um parâmetro largamente utilizado. Espécie de função Define.

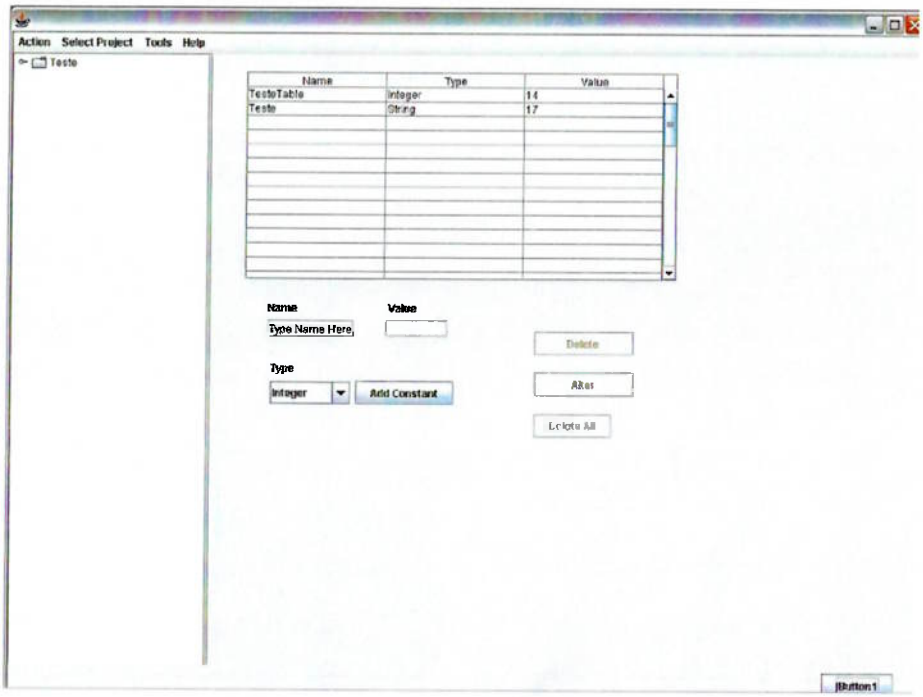


Figura 6 - Tela de cadastro de constantes

7) Cadastro de Tabelas

Usuário descreve lógicamente o banco de dados utilizado como fonte.

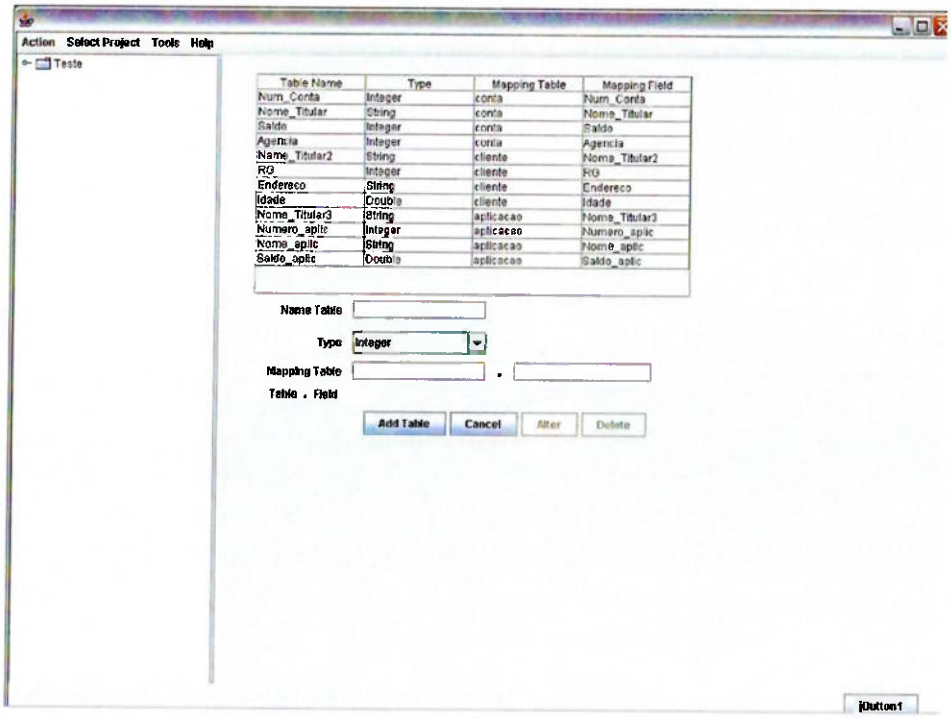


Figura 7 - Tela de cadastro de tabelas

8) Tela de Geração

Permite ao usuário selecionar os cenários a serem considerados na geração da massa de testes.

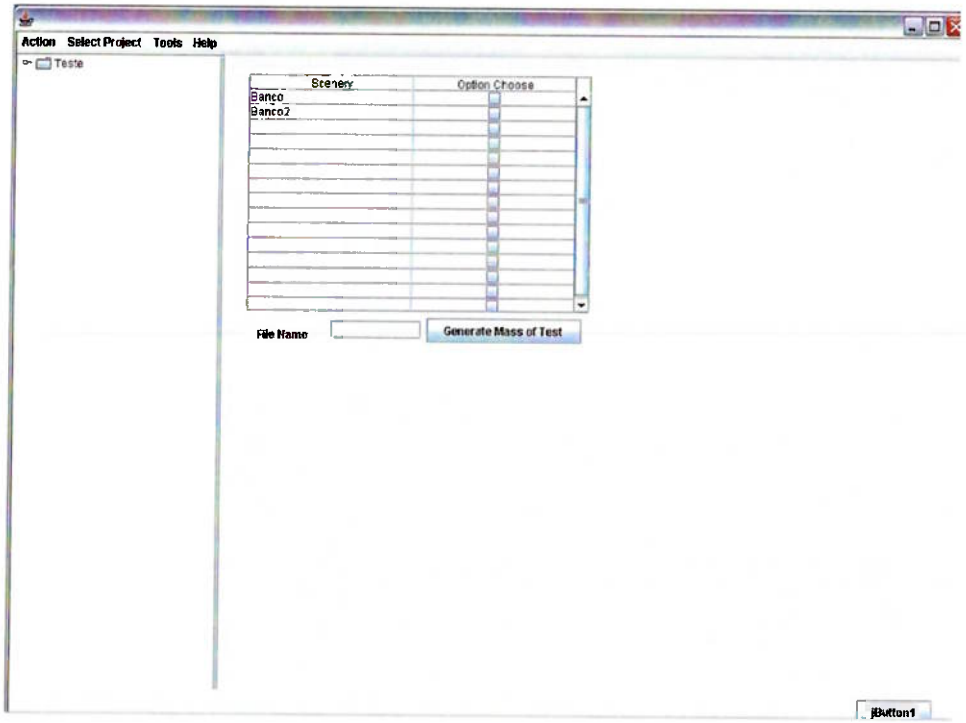


Figura 8 - Tela de Geração

3.3 Modulo de geração de massa de testes

Este é o core do programa. A primeira parte é composta por funções responsáveis pela leitura e quebra das Strings de regras e casos, gerando informações que comporão a consulta. Assim, a regra é quebrada em tabelas, conceitos, valores operações e operadores lógicos sendo gravada em vetores de Strings.

Em seguida, todas as tabelas são reclassificadas no vetor por ordem alfabética e as funções de montagem são chamadas. Essas funções remontam as regras na forma de chamadas sql. Para tanto conectam-se as regras de casos e suas regras personalizadoras através da variável lógica "and".

Com a chamada montada, procedemos com a consulta e os registros encontrados, advindos das regras cadastradas, são gravados em um arquivo txt, que será disponibilizado para o usuário.

Em resumo, o programa tem duas utilidades. A primeira delas é a de orientação na formulação dos cenários de testes. Procura-se automatizar e padronizar o processo, fazendo com que o usuário pense e estude quais cenários são relevantes à sua pesquisa e qual será a estratégia de testes.

Além disso, através de cadastros simples, montam-se consultas complexas na base de dados, facilitando e garantindo que a massa de testes gerada realmente é adequada para aquele determinado caso.

4. Diagramas UML

Apresentam-se aqui os diagramas UML que deram origem ao software, gerados a partir dos casos de uso construídos. Há dois tipos de diagramas: O diagrama de classes, que explicita as classes com seus atributos e métodos, além das interrelações existentes e o diagrama de dados que mostra a lógica e as relações do banco de dados construído para o programa.

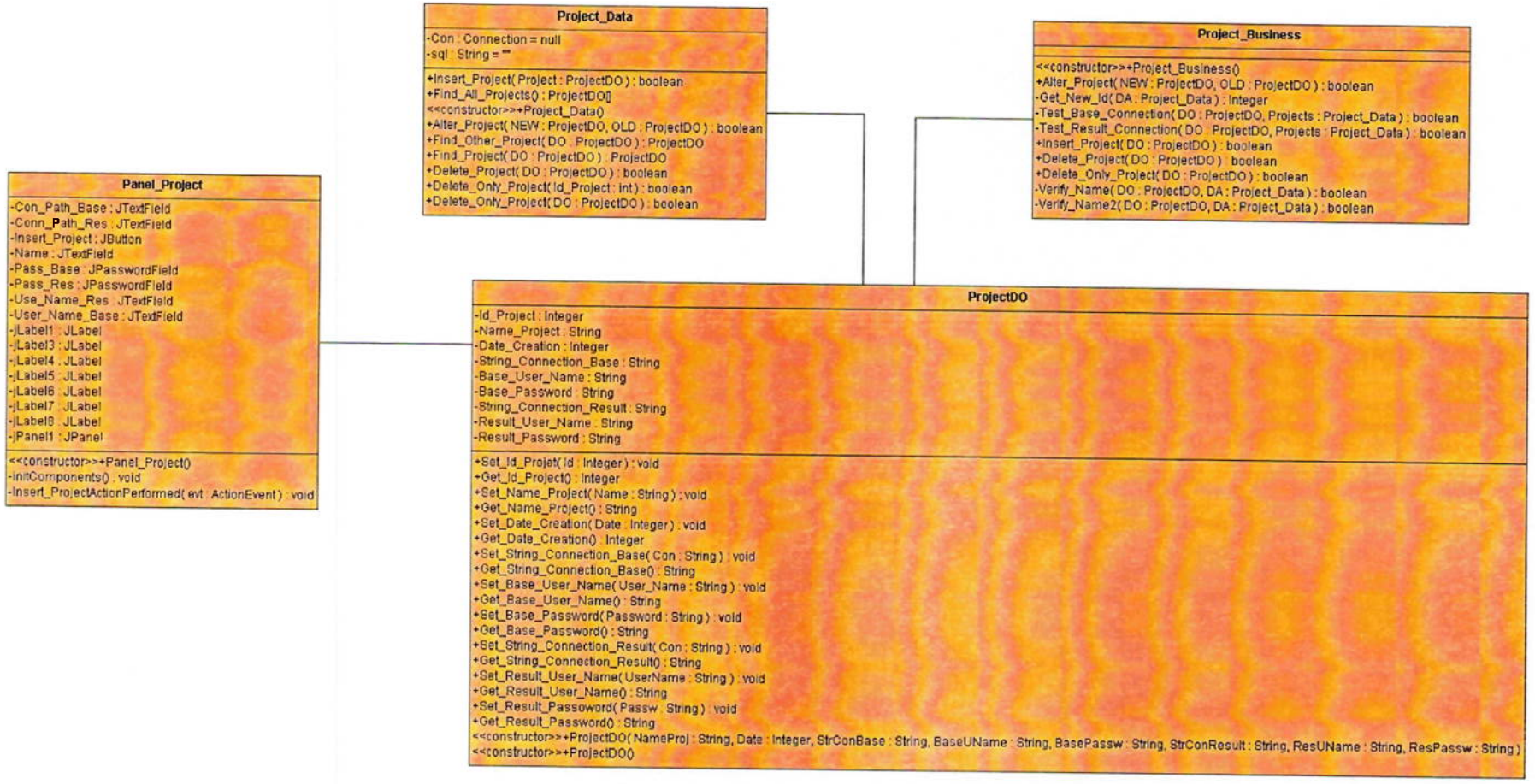


Figura 9 Diagrama UML - Project

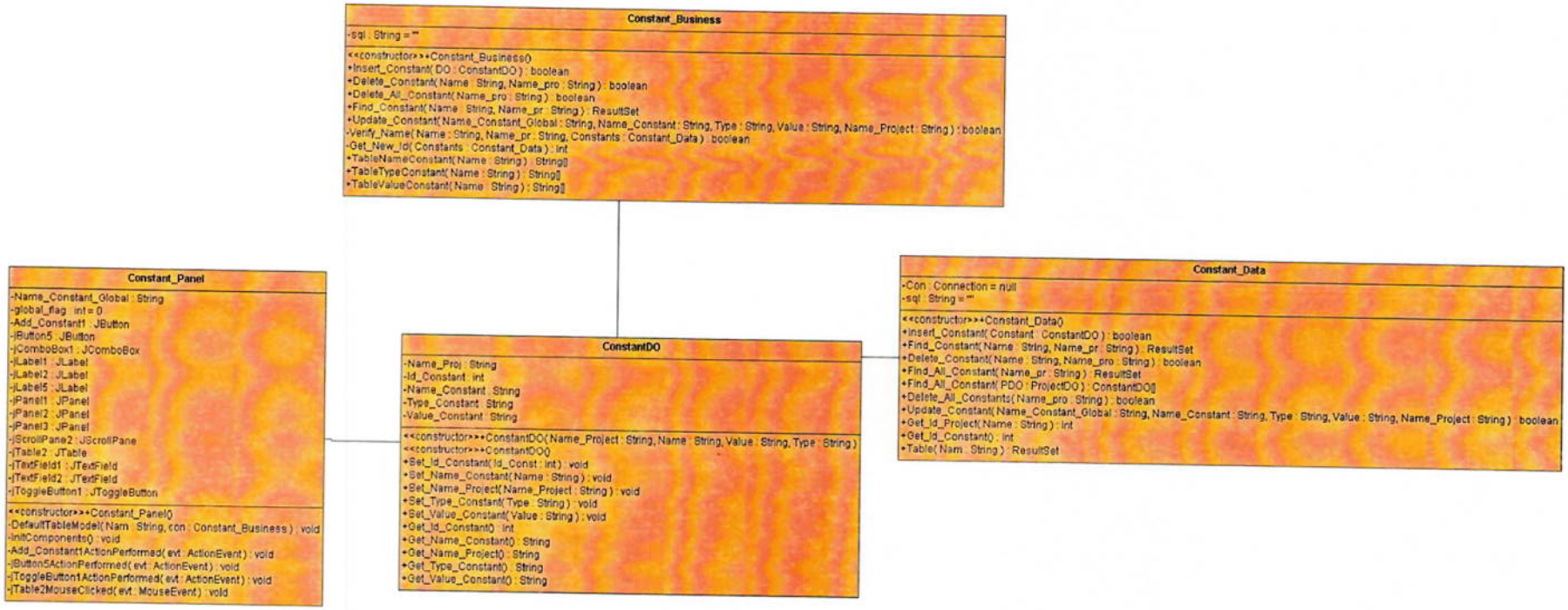


Figura 10 Diagrama UML - Constant

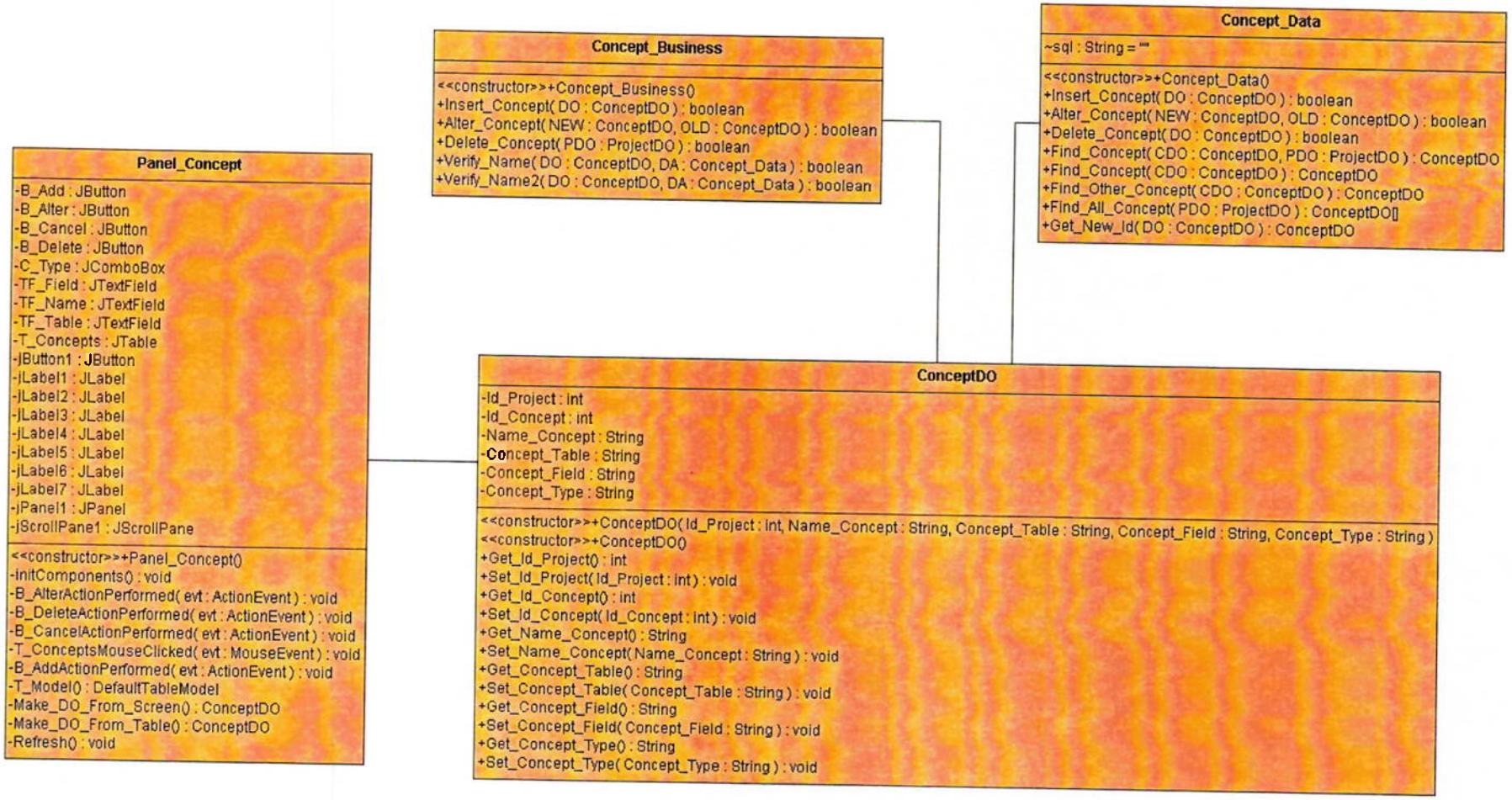


Figura 11 Diagrama UML - Concept

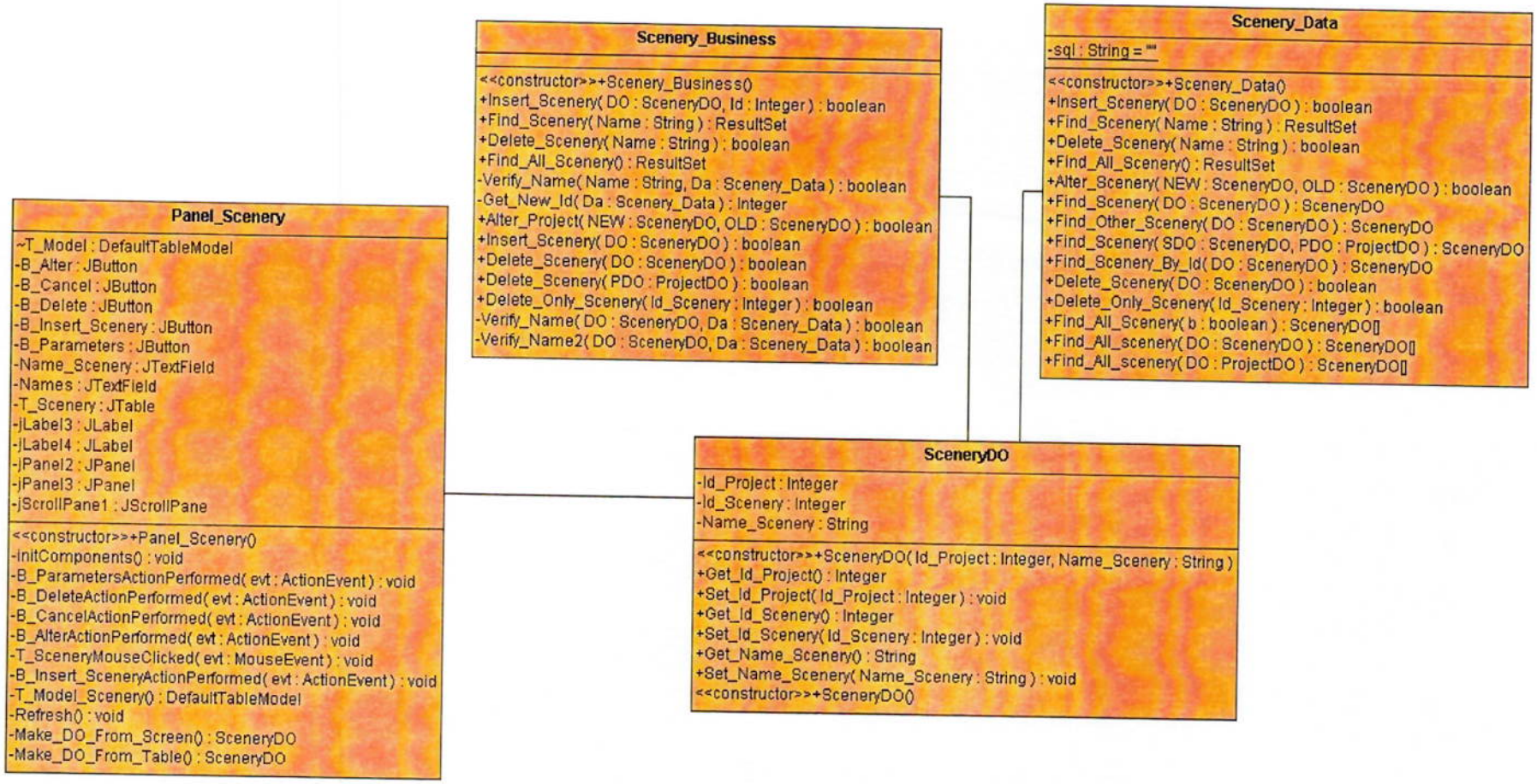


Figura 12 Diagrama UML - Scenery

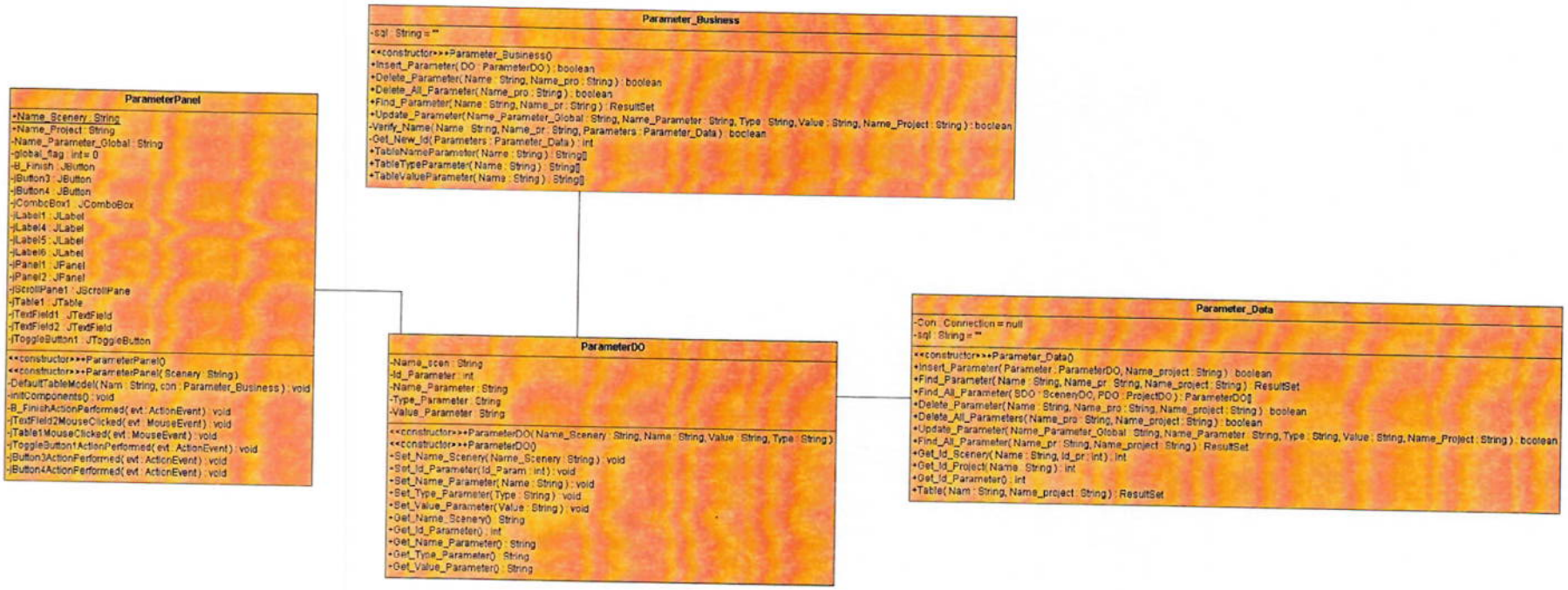


Figura 13 Diagrama UML - Parameter

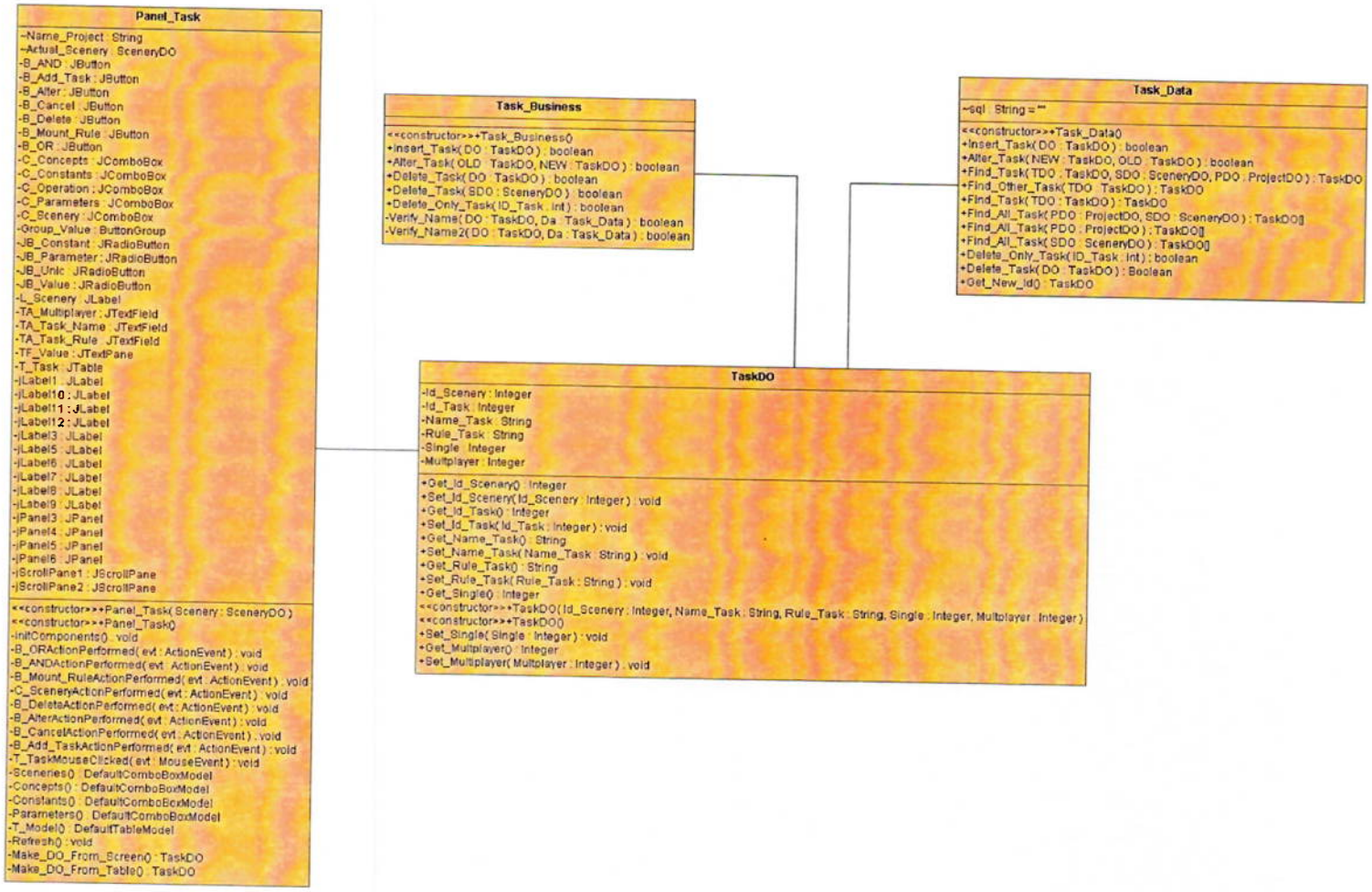


Figura 14 Diagrama UML - Task

Figura 15 Diagrama UML - Rule

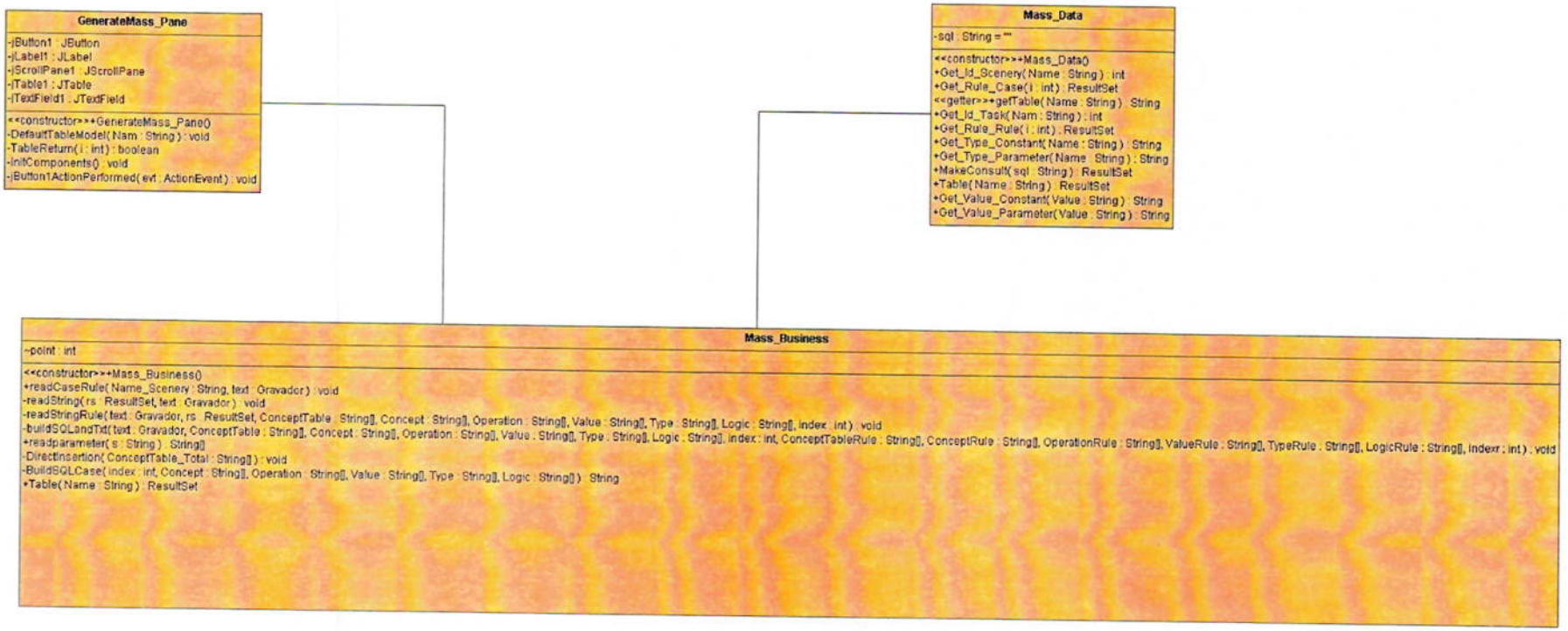


Figura 16 Diagrama UML - Geração da Massa

Figura 17 Diagrama UML – Tela Principal

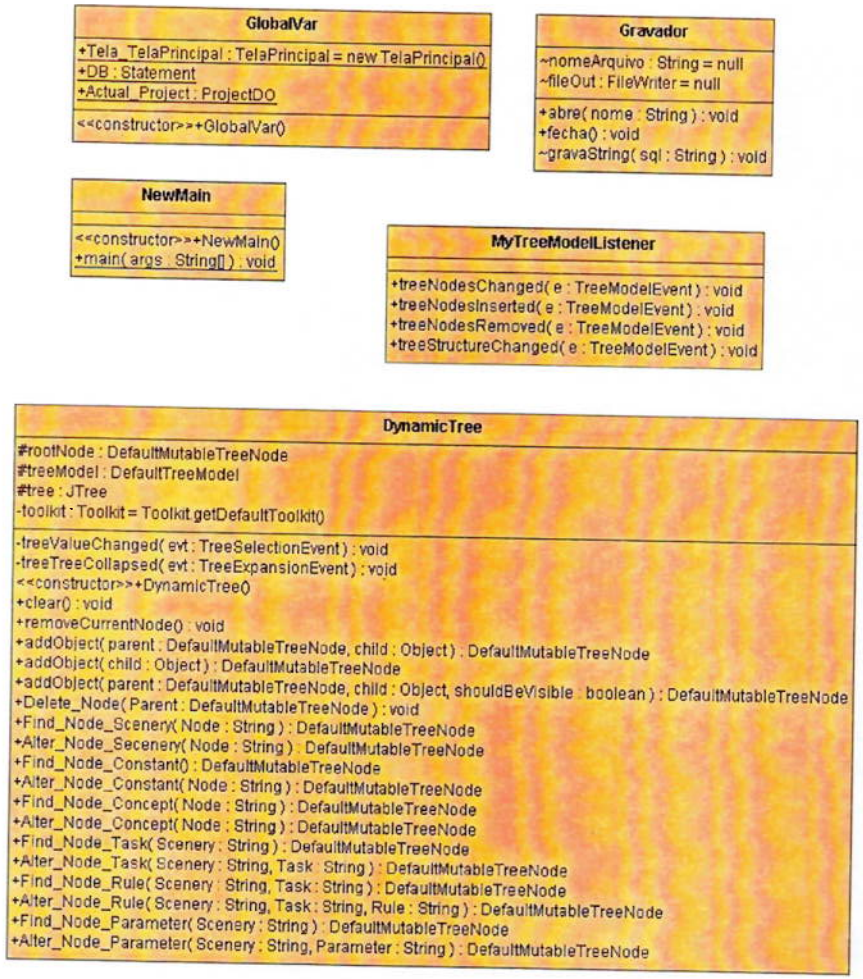


Figura 18 Diagrama UML – Outras Classes

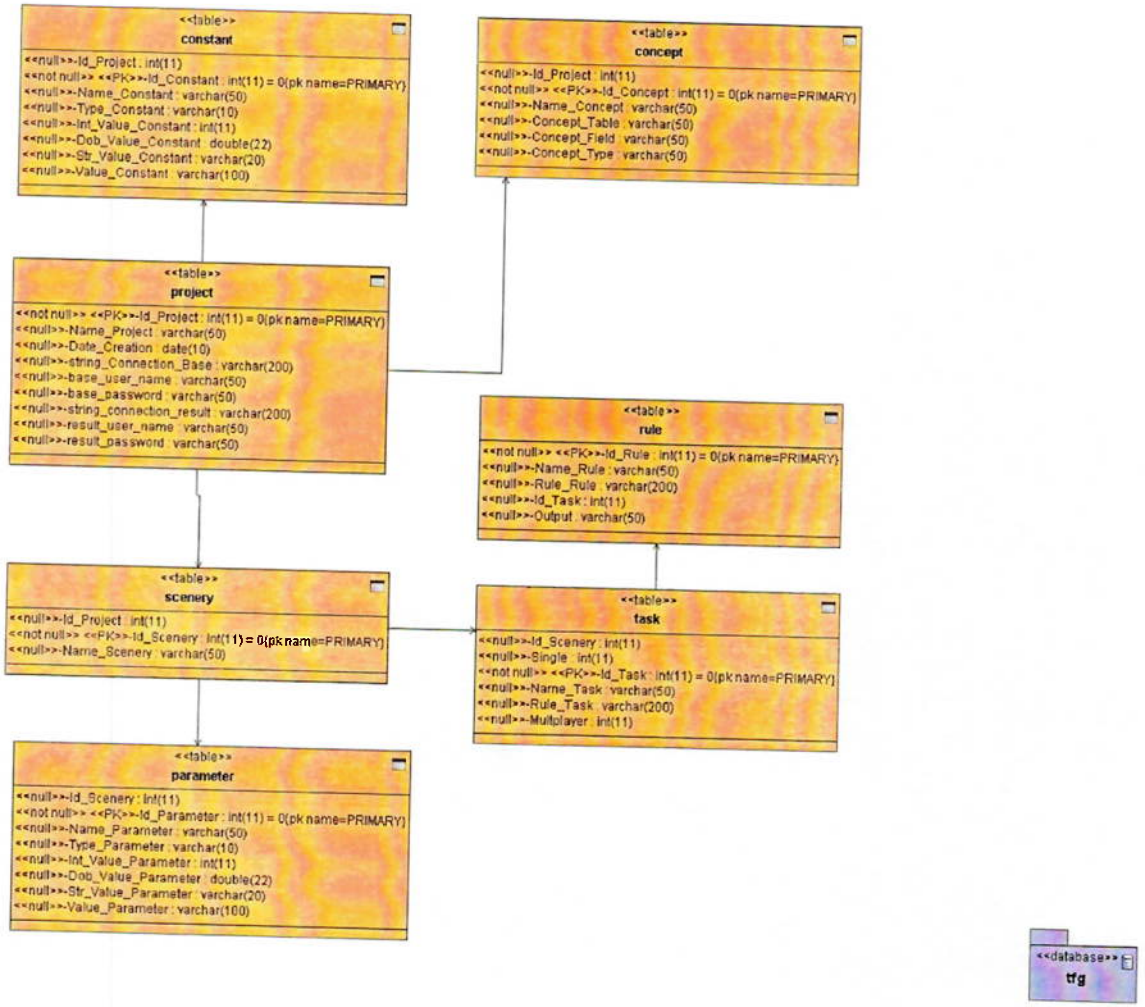


Figura 19 Diagrama UML – Dados

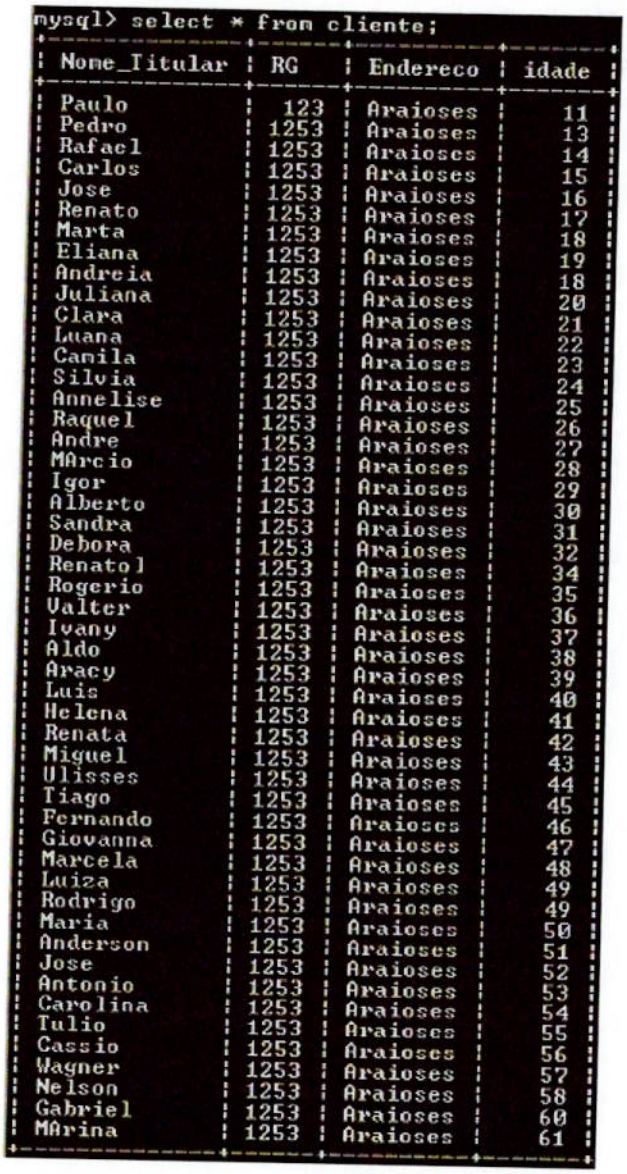
5. Resultados

Como resultado, espera-se que após o cadastro dos cenários, o programa seja capaz de gerar um arquivo txt no formato "CVS", ou seja, exportável para Microsoft Excel, com os registros relevantes para o caso de teste em questão. Para avaliação da eficácia da solução, foi gerada a seguinte base de testes:

```
mysql> describe cliente;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| Nome_Titular | varchar(30)  | YES  |     | NULL    |       |
| RG           | int(10)      | YES  |     | NULL    |       |
| Endereco     | varchar(30)  | YES  |     | NULL    |       |
| idade        | decimal(3,0) | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
```

Figura 20 - Tabela do banco de testes

Esta tabela possui os seguintes registros:



```
mysql> select * from cliente;
+--------------+------+----------+-------+
| Nome_Titular | RG   | Endereco | idade |
+--------------+------+----------+-------+
| Paulo        |  123 | Araioses |    11 |
| Pedro        | 1253 | Araioses |    13 |
| Rafael       | 1253 | Araioses |    14 |
| Carlos       | 1253 | Araioses |    15 |
| Jose         | 1253 | Araioses |    16 |
| Renato       | 1253 | Araioses |    17 |
| Marta        | 1253 | Araioses |    18 |
| Eliana       | 1253 | Araioses |    19 |
| Andreia      | 1253 | Araioses |    18 |
| Juliana      | 1253 | Araioses |    20 |
| Clara        | 1253 | Araioses |    21 |
| Luana        | 1253 | Araioses |    22 |
| Canila       | 1253 | Araioses |    23 |
| Silvia       | 1253 | Araioses |    24 |
| Annelise     | 1253 | Araioses |    25 |
| Raquel       | 1253 | Araioses |    26 |
| Andre        | 1253 | Araioses |    27 |
| MArcio       | 1253 | Araioses |    28 |
| Igor         | 1253 | Araioses |    29 |
| Alberto      | 1253 | Araioses |    30 |
| Sandra       | 1253 | Araioses |    31 |
| Debora       | 1253 | Araioses |    32 |
| Renato1      | 1253 | Araioses |    34 |
| Rogerio      | 1253 | Araioses |    35 |
| Valter       | 1253 | Araioses |    36 |
| Ivany        | 1253 | Araioses |    37 |
| Aldo         | 1253 | Araioses |    38 |
| Aracy        | 1253 | Araioses |    39 |
| Luis         | 1253 | Araioses |    40 |
| Helena       | 1253 | Araioses |    41 |
| Renata       | 1253 | Araioses |    42 |
| Miguel       | 1253 | Araioses |    43 |
| Ulisses      | 1253 | Araioses |    44 |
| Tiago        | 1253 | Araioses |    45 |
| Fernando     | 1253 | Araioses |    46 |
| Giovanna     | 1253 | Araioses |    47 |
| Marcela      | 1253 | Araioses |    48 |
| Luiza        | 1253 | Araioses |    49 |
| Rodrigo      | 1253 | Araioses |    49 |
| Maria        | 1253 | Araioses |    50 |
| Anderson     | 1253 | Araioses |    51 |
| Jose         | 1253 | Araioses |    52 |
| Antonio      | 1253 | Araioses |    53 |
| Carolina     | 1253 | Araioses |    54 |
| Tulio        | 1253 | Araioses |    55 |
| Cassio       | 1253 | Araioses |    56 |
| Wagner       | 1253 | Araioses |    57 |
| Nelson       | 1253 | Araioses |    58 |
| Gabriel      | 1253 | Araioses |    60 |
| MArina       | 1253 | Araioses |    61 |
+--------------+------+----------+-------+
```

Figura 21 – Registros

No programa foram inseridos no caso de teste as seguintes regras:

Cenario: Cliente

Caso: Maior de Idade (Constante cadastrada)

Regra_Caso: Idade > 18;

Regra: RG = 1253 (Parâmetro cadastrado)

O arquivo resultante foi:

Your Case was descripted as:

Select * from cliente where idade > 18 and (RG = 1253)

And the significant registers for this case are:

'Nome_Titular','RG','Endereco','idade',

'Eliana','1253','Araioses','19','Juliana','1253','Araioses','20','Clara','1253','Araioses','21','Luana','1253','Araioses','22','Camila','1253','Araioses','23','Silvia','1253','Araioses','24','Annelise','1253','Araioses','25','Raquel','1253','Araioses','26','Andre','1253','Araioses','27','MArcio','1253','Araioses','28','Igor','1253','Araioses','29','Alberto','1253','Araioses','30','Sandra','1253','Araioses','31','Debora','1253','Araioses','32','Renato]','1253','Araioses','34','Rogerio','1253','Araioses','35','Valter','1253','Araioses','36','Ivany','1253','Araioses','37','Aldo','1253','Araioses','38','Aracy','1253','Araioses','39','Luis','1253','Araioses','40','Helena','1253','Araioses','41','Renata','1253','Araioses','42','Miguel','1253','Araioses','43','Ulisses','1253','Araioses','44','Tiago','1253','Araioses','45','Fernando','1253','Araioses','46','Giovanna','1253','Araioses','47','Marcela','1253','Araioses','48','Luiza','1253','Araioses','49','Rodrigo','1253','Araioses','49','Maria','1253','Araioses','50','Anderson','1253','Araioses','51','Jose','1253','Araioses','52','Antonio','1253','Araioses','53','Carolina','1253','Araioses','54','Tulio','1253','Araioses','55','Cassio','1253','Araioses','56','Wagner','1253','Araioses','57','Nelson','1253','Araioses','58','Gabriel','1253','Araioses','60','MArina','1253','Araioses','61'

Em seguida foram inseridos:

Cenario: Cliente
Caso: Meia Idade
Regra_Caso: Idade > 30 (Valor inserido);
Regra: Nome_Titular = Tiago (Constante cadastrada)

O arquivo resultante foi:

Your Case was descripted as:

Select * from cliente where idade > 30 and Nome_Titular = 'Tiago'

And the significant registers for this case are:

'Nome_Titular','RG','Endereco','idade',
'Tiago','1253','Araioses','45',

6. Conclusão

Em vista da escassez de material sobre o tema, acredita-se no pionerismo deste trabalho, bem como na aplicabilidade nos mais variados casos de teste. Bem utilizado, o programa pode apresentar ganho para o testador, tanto em termos de tempo quanto em termos de eficiência. Deixa-se como sugestão para melhorias futuras, a saída do programa como uma planilha Excel ou apenas os ID dos registros, para que o usuário possa se conectar em seu banco de dados pelo próprio programa e buscar os registros. Além disso, pode-se mapear automaticamente o banco de dados do usuário, incluindo as relações entre tabelas, fazendo com que se possa incrementar ainda mais a complexidade das consultas sql.

7. Bibliografia

BARRETTO, M. P., **Notas de Aula – PMR2490**, São Paulo, 2005

BEIZER, Boris. **Software testing techniques**. 2nd ed. New York : Van Nostrand Reinhold, 1990

DE MILO, **Richard A. Software testing and evaluation.** Menlo Park : The Benjamin Cummings Publishin Company, Inc, 1987

HETZEL, William C.**The complete guide to software testing.** 2nd ed. Mass. : QED Information Sciences, 1988

KANER, Cem. **Lessons learned in software testing : a context-driven approach.** New York : Wiley, 2002

MILLER, Edward. **Software testing & validation techniques : tutorial.** 2. Ed. New York : Ieee, 1981

MYERS, Glenford J.**The art of software testing New York**. New York: Editora Wiley. 1979

PERRY, William E. **Effective methods for software testing.** New York : Wiley, 1995

PERRY, William E. **Effective methods for software testing.** 2nd ed. New York : Wiley, 2000

POSTON, Robert M. **Automating specification-based software testing.** Los Alamitos, Calif. : IEEE Computer Society Press, 1996

PRESSMAN, R. S.. **Engenharia de Software**. São Paulo: Bookmark, 2005

ROYER, Thomas C. **Software testing management : lif on the critical path.** Englewood Cliffs : P T R Prentice Hall, 1993

SIEGEL, Shel. **Object oriented software testing : a hierarchical approach.** New York : Wiley Computer Pub., 1996

Apêndice 1

USP::Poli::Mecatrônica/PMR2550

Versão 4.*0*

Histórico das Revisões

Data	Versão	Descrição	Autor
15/06	1.0	Descrição dos casos de Uso após o detalhamento do mesmo.	Renato Albolea Rodrigo Papetti
13/07	2.0	Revisão dos Casos de Uso, após reunião com professor Marcos Barretto	Renato Albolea Rodrigo Papetti
07/09	3.0	Revisão dos Casos de Uso, após implementação parcial do protótipo	Renato Albolea Rodrigo Papetti
20/11	4.0	Revisão dos Casos de Uso após implementação e testes do Protótipo	Renato Albolea Rodrigo Papetti

Índice

Índice de Figura

1 Apresentação

1.1 Objetivo

O objetivo deste documento é a especificação de requisitos para o conjunto de funcionalidades referido como *Sistema de Geração de Massa de Teste*. Nesse documento serão detalhados os casos de uso referentes à criação do Data Generator.

Este sistema irá, a partir de alguns parâmetros fornecidos pelo usuário, indicar quais registros, presentes num banco de dados previamente fornecido, são relevantes para os testes que se seguirão. Assim, o usuário entra com o cenário de testes, o desenho do banco de dados e outros parâmetros, e recebe um arquivo txt com os registros que devem ser usados daquele banco.

2 Conceitos Gerais

2.1 Dicionário de conceitos

Abaixo serão apresentados alguns conceitos que são usados no programa de geração de massa de teste.

- Projeto: No programa é entendido como projeto (Project) um conjunto determinado de tabelas que constituem uma base de dados que será avaliada para a geração da massa de teste.

- Cenário: No programa é entendido como cenário (Scenery) um conjunto de regras, constantes e parâmetros que descrevem de forma macro o caso de teste.

- Caso: No programa é entendido como casos (Case) uma particularização do universo de registros da base de dados que atende a algumas características.

- Regra: No programa é entendido como regras (Rule) uma particularização do universo de registro selecionados no Caso ao qual a Regra pertence.

- Conceito: No Programa é entendido como Conceito (Concept) o campo de uma tabela de um banco de dados

2.2 Tecnologia Utilizada

O servidor de banco de dados que será utilizado para o desenvolvimento e teste do sistema em questão será o MySQL, sendo a interface desenvolvida em linguagem Java através da IDE NetBeans 5.0

2.3 Sistema de Cadastro

O sistema de cadastro é caracterizado por possuir um conjunto de telas que permitirão ao usuário construir todo o cenário em que se desenvolve o teste. Será constituido por n telas construídas para que se possa descrever o banco de dados, como mapear seus registros,quais as principais constantes e parâmetros e, principalmente, quais os casos e as regras do teste. Neste sistema, localizam-se todas as telas de interface com o usuário.

2.4 Sistema de Geração

O sistema de consulta é caracterizado por possuir toda a inteligência do programa. Após todos os dados serm cadastrados de forma bem sucedida, o sistema usa as informações para gerar, automaticamente, consultas sql que procurarão, dentre todos os regsitros presentes na base de dados, aqueles que se adequam aos casos de testes que o usuário cadastrou no sistema.

3 Adicionar Projeto – GMT001

3.1 Breve Descrição

Destina-se ao cadastramento de um novo projeto.

3.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

3.3 Pré-Condições

O ator deve ter selecionado a opção New Project na aba Action.

3.4 Fluxo de Eventos

Não se aplica.

3.4.1 Fluxo Básico

- O Sistema exibe a tela "New Project" com os campos:
  - ✓ Project Name
  - ✓ Connection path to the base
  - ✓ User name of the base
  - ✓ Password of the base
  - ✓ Connection path to the result base
  - ✓ User name of the result base
  - ✓ Password of the result base

- Quando usuário clicar em Create Project o sistema verifica se o nome do projeto é único e se as strings de conexão fornecidas são validas;
- Se nome do projeto é único e se as conexões com os bancos de dados são validas o sistema grava os dados e constrói uma arvore que contem os elementos relacionados ao projeto conforme se vê na figura abaixo;
- Se não sistema exibe uma mensagem pedindo ao usuário para entrar com um nome novo ou corrigir a string de conexão;

A Tela new project é a seguinte:

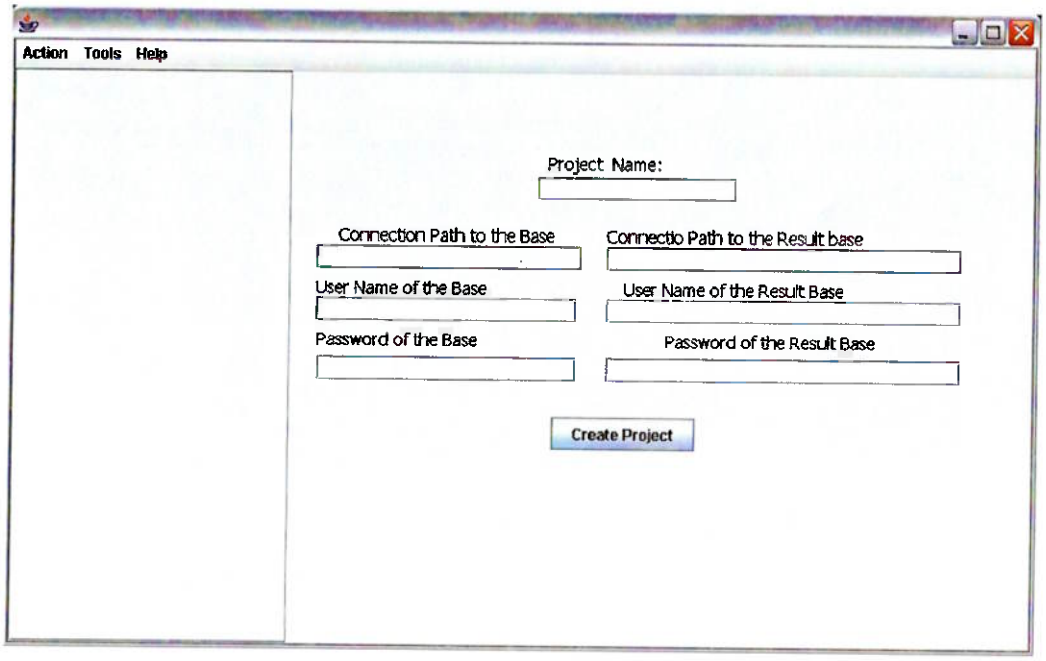


Figura 3.1 Tela de cadastro de Projeto(New Project)

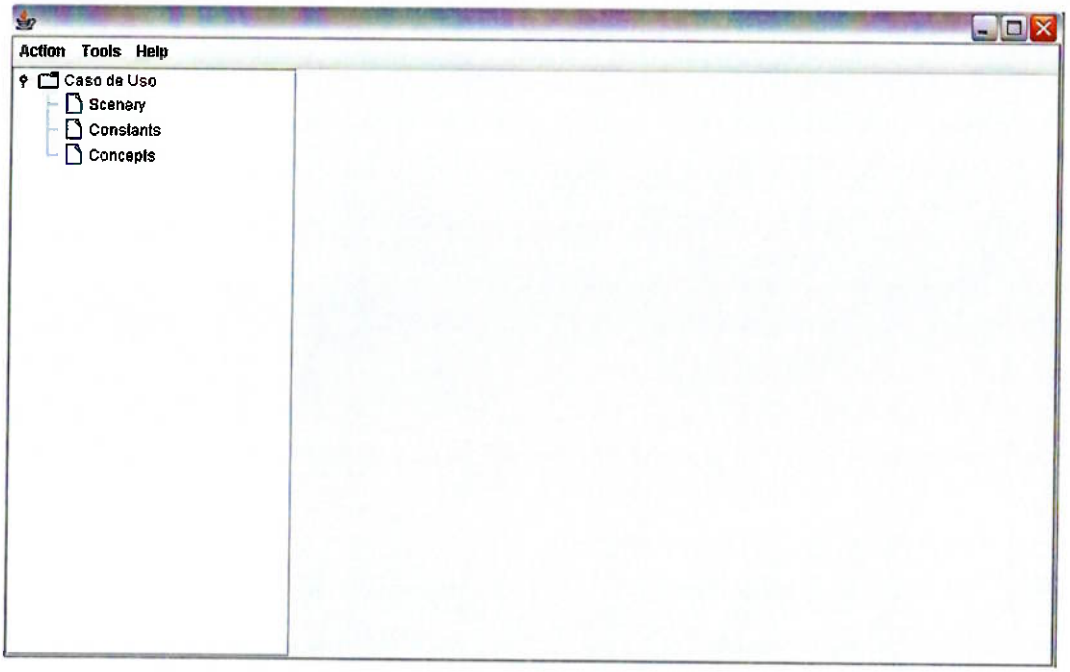


Figura 3.2 Árvore do Projeto

3.4.2 Fluxos Alternativos

Não aplicavel

3.4.3 Requerimentos Especiais

Não aplicável.

3.4.4 Pós-Condições

Após a criação do novo projeto, qualquer outro projeto que estiver aberto será fechado automaticamente.

3.4.5 Pontos de Extensão

Não Aplicável.

4 Adicionar, Alterar, Excluir Constantes – GMT002

4.1 Breve Descrição

Destina-se a permitir o cadastro de constantes por parte do usuário.

4.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

4.3 Pré-Condições

Para que se possa cadastrar uma constante é necessário haver um projeto corretamente cadastrado.

4.4 Fluxo de Eventos

Não se aplica.

4.4.1 Fluxo Básico

- O sistema exibe a tela de cadastro de constantes com os botões "Alter" e "Delete" desabilitados e os seguintes campos:
  - ✓ Name
  - ✓ Value
  - ✓ Type (Esse campo só pode assumir os valores Integer, Double e String)

- Quando o usuário clica no botão "Add Constant" o sistema verifica se o nome da constante é único e se o tipo determinado pelo usurário é valido;
- Se o nome for único e o valor fornecido for valido o sistema atualiza a arvore do programa e grava os dados;
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome ou informando que o tipo da constante selecionado não é compatível com o valor fornecido;

A tela de cadastro de constante é a seguinte:

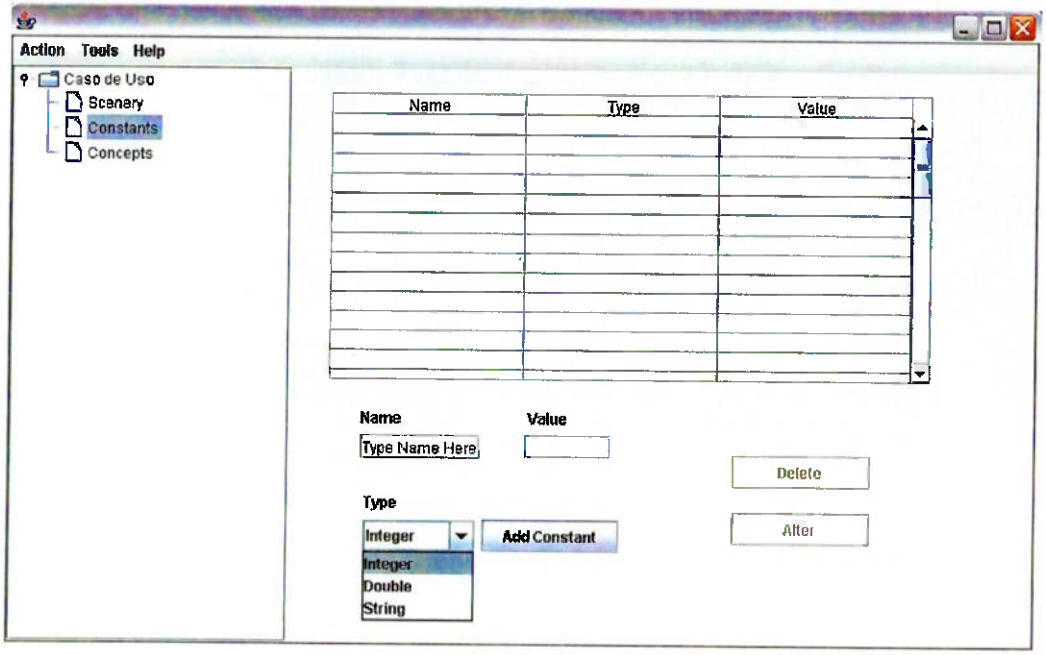


Figura 4.1 Tela de cadastro de Constante(Add Constant)

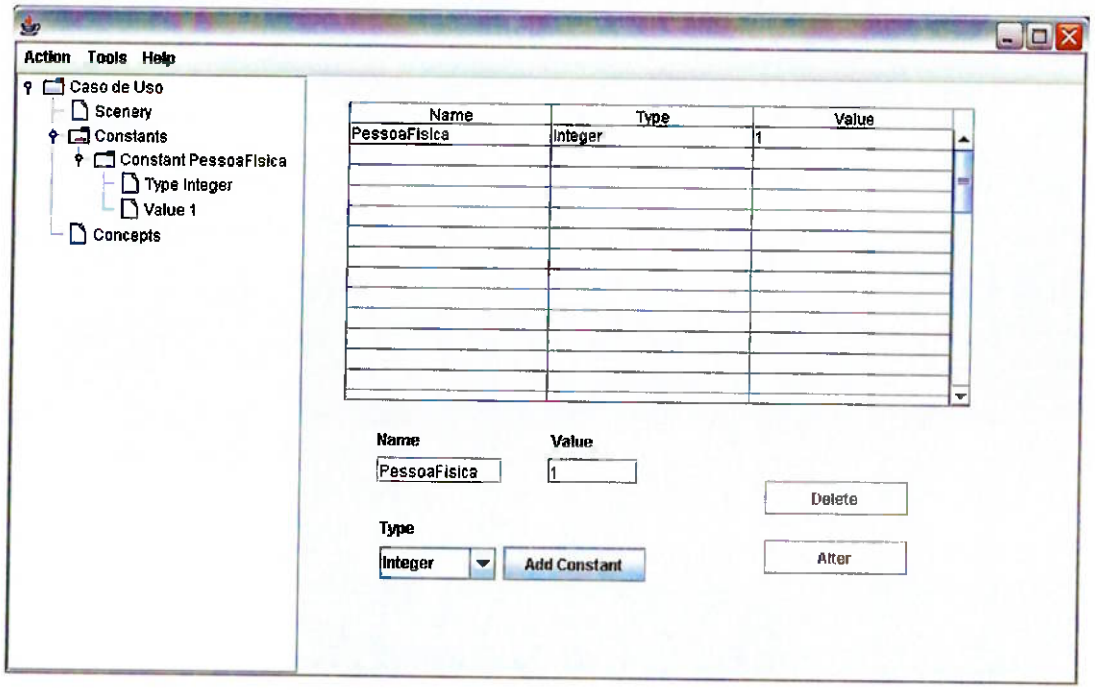


Figura 4.2 Tela após a ainclusão da Constante

4.4.2 Fluxos Alternativos

A)Exclusão de uma constante

- O ator seleciona uma constante já cadastrada com na tabela de constantes
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre a constante são copiadas para os seus respectivos campos de criação;
- Se o ator clicar no botão "Delete" o sistema exibe uma mensagem padrão pedindo a confirmação da exclusão;
- Se o ator confirmar a exclusão a tela é atualizada;

- Se o ator não confirmar a exclusão o processo é abortado e nenhuma modificação é feita;

B)Alteração

- O ator seleciona uma constante já cadastrada com na tabela de constantes
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre a constante são copiadas para os seus respectivos campos de criação;
- Quando o usuário clica no botão "Alter" o sistema verifica se o nome da constante é único e se o tipo determinado pelo usurário é valido;
- Se o nome for único e o valor fornecido for valido o sistema atualiza a arvore do programa e grava os dados;
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;
- Se ator clicar no botão "Cancel" a operação de alteração é abortada e nenhuma modificação é feita.

4.4.3 Requerimentos Especiais

Não se aplica.

4.4.4 Pós-Condições

Não se aplica.

4.4.5 Pontos de Extensão

Nenhum.

5 *Adicionar, Alterar, Excluir Conceito– GMT003*

5.1 Breve Descrição

Destina-se ao cadastro de conceitos ao programa para que seja criada uma réplica da estrutura do banco de dados do usuário.

5.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

5.3 Pré-Condições

Para que se possa cadastrar um conceito é necessário haver um projeto corretamente cadastrado.

5.4 Fluxo de Eventos

Não se aplica.

5.4.1 Fluxo Básico

- O sistema exibe a tela de cadastro de Conceito com os botões "Alter" e "Delete" desabilitados e com os seguintes campos:
  - ✓ Concept Name

- ✓ Type(Esse campo só pode assumir os valores Integer, Double e String)
- ✓ Mapping: Table
- ✓ Mapping: Field

- Quando o usuário clica no botão "Add Concept" o sistema verifica se o nome do conceito é único no projeto;
- Se o nome for único o sistema atualiza a tela;
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;

A tela de criação de conceitos tem a seguinte forma:

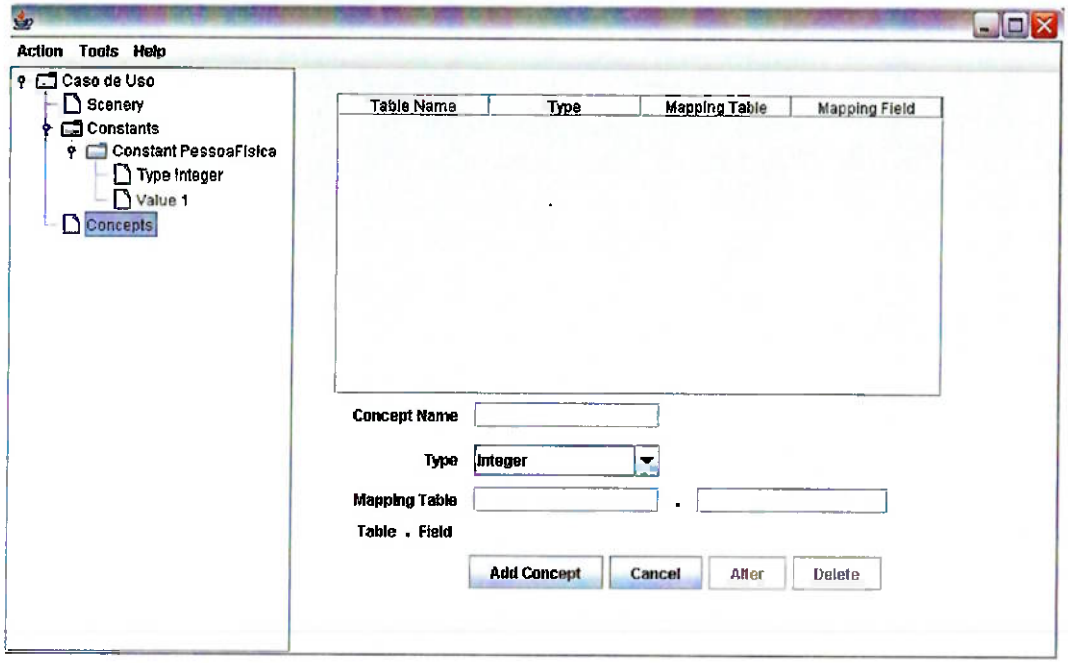


Figura 5.1 Tela de cadastro de Conceito(Add Concept)

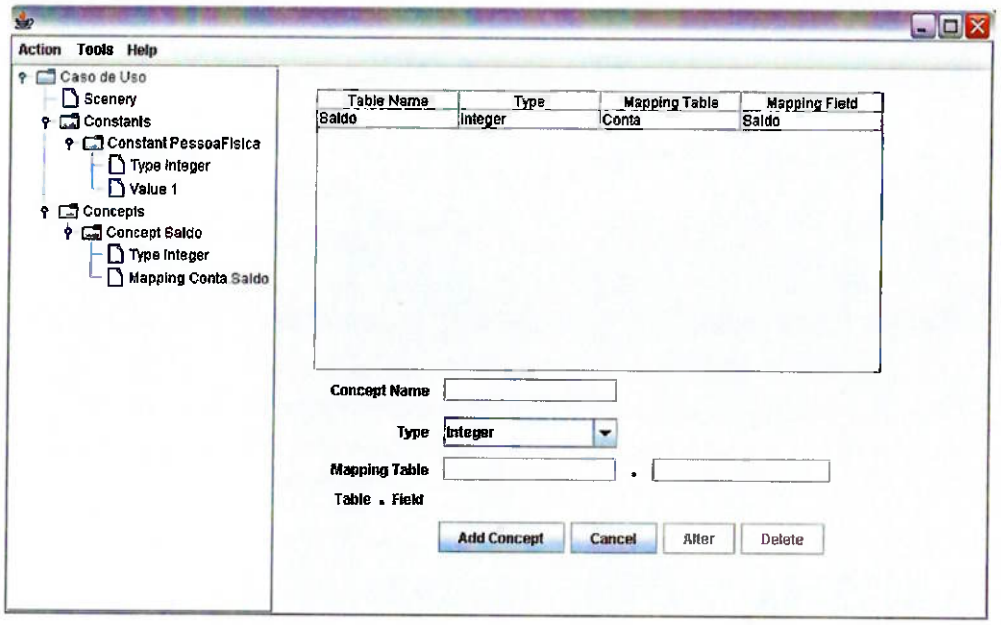


Figura 5.2 Tela após a inclusão de um Conceito

5.4.2 Fluxos Alternativos

A)Exclusão de um conceito

- O ator seleciona um conceito já cadastrado na tabela de Conceitos
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre o conceito são copiadas para os seus respectivos campos de criação;
- Se o ator clicar no botão "Delete" o sistema exibe uma mensagem padrão pedindo a confirmação da exclusão;
- Se o ator confirmar a exclusão a tela é atualizada;
- Se o ator não confirmar a exclusão o processo é abortado e nenhuma modificação é feita;

B)Alteração

- O ator seleciona um conceito já cadastrado com um duplo clique na tela "Concept Creation";
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre o conceito são copiadas para os seus respectivos campos de criação;
- Quando o usuário clica no botão "Alter" o sistema verifica se o nome do conceito é único no projeto;
- Se o nome for único o sistema atualiza a tela "Concept Creation"
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;
- Se ator clicar no botão "Cancel" a operação de alteração é abortada e nenhuma modificação é feita;

5.4.3 Requerimentos Especiais

Não aplicável.

5.4.4 Pós-Condições

Não se aplica.

5.4.5 Pontos de Extensão

Não Há.

6 Adicionar, Alterar, Excluir Cenário – GMT005

6.1 Breve Descrição

Destina-se ao cadastro de novos cenários de testes.

6.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

6.3 Pré-Condições

Para que se possa cadastrar um conceito é necessário haver um projeto corretamente cadastrado.

6.4 Fluxo de Eventos

Não se aplica.

6.4.1 Fluxo Básico

- O sistema mostra a primeira tela do wizard de criação de cenários com os botões "Alter", "Delete" e "View Parameters" desabilitados e com o campo:
  - ✓ Scenary Name
- Quando o usuário clicar no botão "Add Scenary" o sistema verifica se o nome do cenário é único no projeto;

- Se o nome for único o sistema grava os dados, atualiza a arvore do projeto e mostra a segunda tela do Wizard que é a tela de cadastro de Parametro;
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;

A tela de cadastro de Cenário é a seguinte:

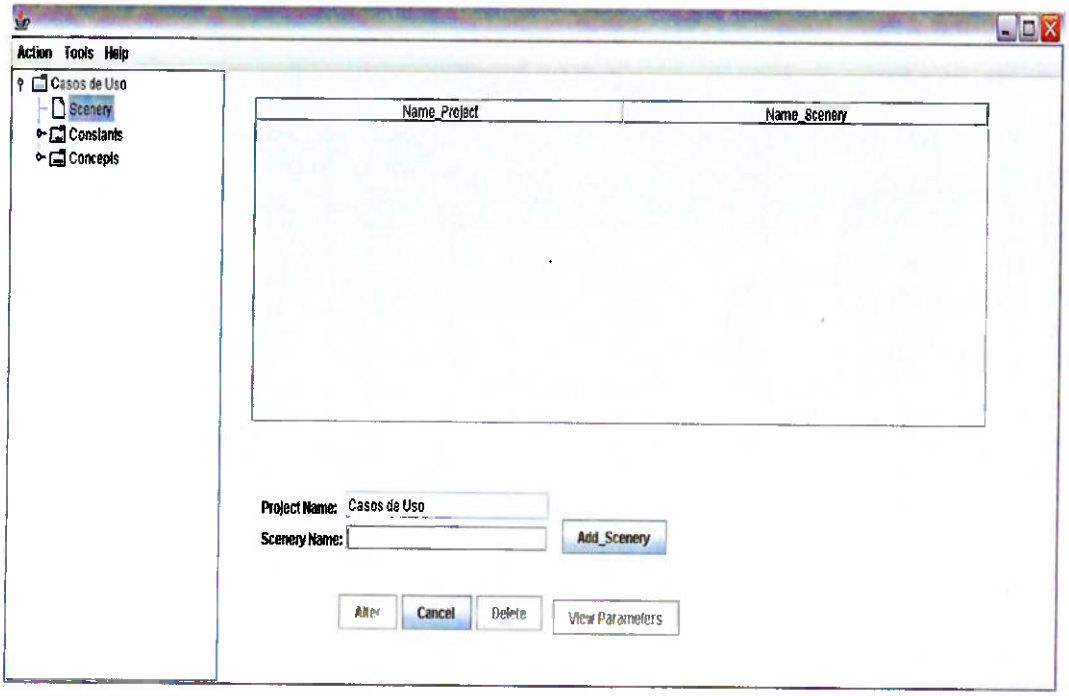


Figura 6.1 Tela de cadastro de Cenário(Add Scenery)

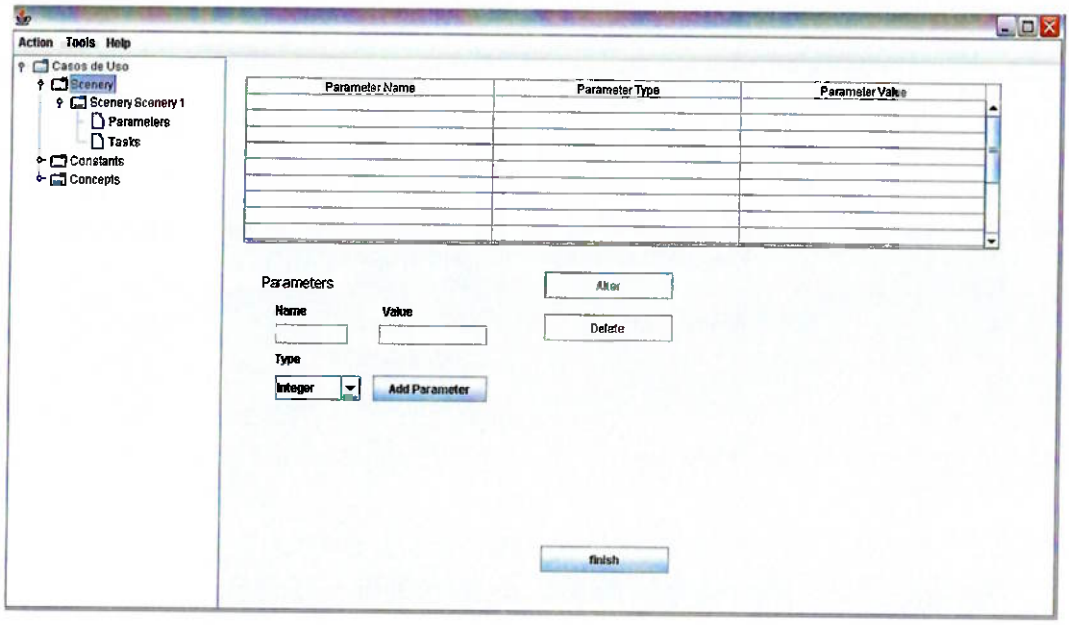


Figura 6.2 Tela após o cadastro de um cenário

6.4.2 Fluxos Alternativos

A)Exclusão de um Cenario

- O ator seleciona um cenario já cadastrado na tela "Add Scenary";
- O Sistema habilita os botões "Alter", "Delete" e "View Parameters"
- O Sistema copia o nome do cenário para o campo "Name Scenery";
- Se o ator clicar no botão "Delete" o sistema exibe uma mensagem padrão pedindo a confirmação da exclusão;
- Se o ator confirmar a exclusão a tela "Add Scenary" é atualizada e os dados são gravados;
- Se o ator não confirmar a exclusão o processo é abortado e nenhuma modificação é feita;

B)Alteração

- O ator seleciona um cenário já cadastrado na tela "Add Scenary";
- O Sistema habilita os botões "Alter", "Delete" e "View Parameters"
- O Sistema copia o nome do cenário para o campo "Name Scenery";
- Quando o usuário clicar no botão "Alter" o sistema verifica
- se o nome é único no projeto
- Se for unico o sistema atualiza a tela de cadastro de Cenário, grava os dados e abra e a tela seguinte do wizard(tela de cadastro de Parâmetro)
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;

C)Mostrar Parâmetros

- O ator seleciona um cenário já cadastrado na tela "Add Scenary";
- O Sistema habilita os botões "Alter", "Delete" e "Show Parameter"
- O Sistema copia o nome do cenário para o campo "Name Scenery";
- Quando o usuário clicar no botão "View Parameters" o sistema abre a tela de cadastro de Parâmetros

Se ator clicar no botão "Cancel" a operação de alteração é abortada e nenhuma modificação é feita;

6.4.3 Requerimentos Especiais

Não aplicável.

6.4.4 Pós-Condições

Após o cadastramento do Cenário deve-se abrir a segunda tela do wizard, relativa a criação de parâmetros, descrita no próximo caso de uso.

6.4.5 Pontos de Extensão

Não se aplica.

7 Adicionar, Alterar e Excluir Parâmetro - GMT006

7.1 Breve Descrição

Destina-se a cadastrar parâmetros aos cenários criados na tela Wizard "Add Scenary";

7.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

7.3 Pré-Condições

É necessário ter um cenário previamente cadastrado.

7.4 Fluxo de Eventos

Não se aplica.

7.4.1 Fluxo Básico

- O sistema mostra a tela de criação de parâmetros com os botões "Alter" e "Delete" desabilitados e com os campos:
  - ✓ Name
  - ✓ Type(Esse campo só pode assumir os valores Integer, Double e String)
  - ✓ Values(Os valore devem ser inseridos com espaços e sem virgulas)

- Quando o usuário clicar no botão "Add Parameter" o sistema verifica se o nome do parâmetro é único no Cenario e se os valores fornecidos são condizentes com o tipo selecionado;
- Se o nome for único o sistema grava os dados e atualiza a arvore do projeto e a tela "Add Parameter";
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;
- Quando o usuário clicar no botão "Finish" a tela é encerrada voltando para a tela "Add Scenery";

A tela de cadastro de parâmetro("Add Parameter") é a seguinte

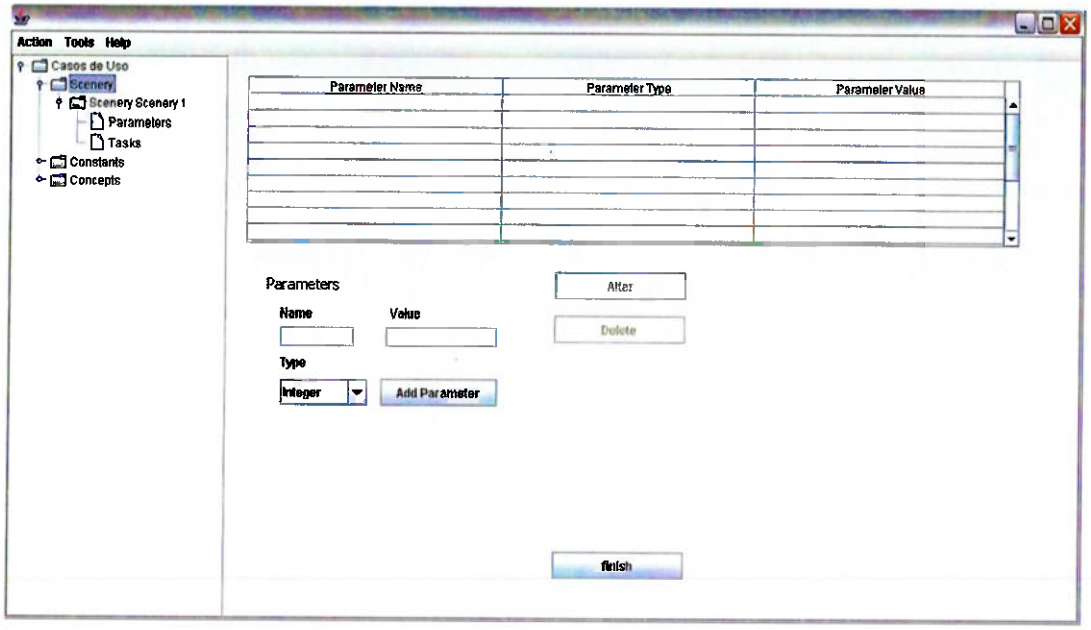


Figura 7.1 Tela de cadastro de Parâmetro(Add Parameter)

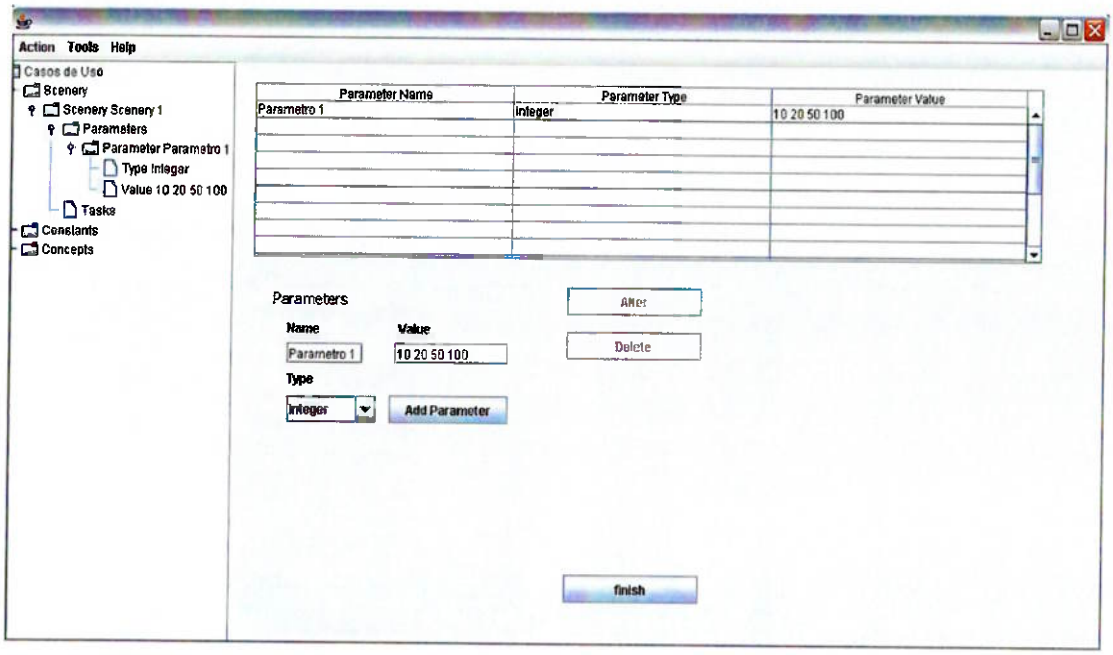


Figura 7.2 Tela após o cadastro de um Parâmetro

7.4.2 Fluxos Alternativos

A)Exclusão de um parâmetro

- O ator seleciona um parâmetro já cadastrado na tela "Add Parameter";
- O sistema habilita os botões "Alter" e "Delete" e copia os dados do parâmetro selecionado para os seus respectivos campos;
- Se o ator clicar no botão "Delete" o sistema exibe uma mensagem padrão pedindo a confirmação da exclusão;
- Se o ator confirmar a exclusão a tela "Add Parameter" é atualizada e os dados são gravados;

- Se o ator não confirmar a exclusão o processo é abortado e nenhuma modificação é feita;

B)Alteração

- O ator seleciona um cenário já cadastrado na tela "Add Parameter";
- O sistema habilita os botões "Alter" e "Delete" e copia os dados do parâmetro selecionado para os seus respectivos campos;
- Quando o usuário clicar no botão "Accept Alter" o sistema verifica se o nome é único no cenário;
- Se for único o sistema atualiza a arvore do projeto tela "Add Parameter" e grava os dados
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;

7.4.3 Requerimentos Especiais

Não aplicável.

7.4.4 Pós-Condições

Não aplicável.

7.4.5 Pontos de Extensão

Não aplicável.

8 Adicionar, Alterar, Excluir Caso – GMT008

8.1 Breve Descrição

Destina-se ao cadastro de casos aos cenários já criados.

8.2 Atores

Este caso de uso é de uso de todos os usuários do sistema

8.3 Pré-Condições

O ator deve ter criado um cenário para poder adicionar casos a ele.

8.4 Fluxo de Eventos

Não se aplica.

8.4.1 Fluxo Básico

- O sistema mostra a tela de criação de casos com os botões "Alter" e "Delete" desabilitado e os campos:
  - ✓ Concept(Check Box com todos os Conceitos cadastrados)
  - ✓ Operation(Pode assumir os valores =, <, >, !=)
  - ✓ Value
  - ✓ Parameter(Check Box com todos os Parâmetros cadastrados naquele Cenário)

- ✓ Constant(Check Box com todos as Constantes cadastrados no projeto)
- ✓ Unic
- ✓ Multiplayer
- ✓ Case Name
- ✓ Rule

- Quando o usuário clicar no botão "Mount Rule" o sistema escreve no campo Rule uma regra conforme os itens selecionados pelo usuário da seguinte forma:
    1. Copia-se para uma String o nome do conceito selecionado com um espaço em branco no final do nome;
    2. Soma-se a String a operação selecionada com um espaço em branco no final;
    3. Verifica-se qual tipo de valor foi selecionado(Value, Constant ou Parameter)
        - Se for Value soma-se a string um v minúsculo com um espaço em branco e soma-se o valor do campo value;
        - Se for Constant soma-se a string um c minúsculo com um espaço em branco e soma-se o nome da constante selecionada;
        - Se for Parameter soma-se a string um p minúsculo com um espaço em branco e soma-se o nome do parâmetro selecionado;
    4. Soma-se a String resultante a String que já contiver no campo Rule
    5. Se quiser adicionar mais uma regra ao mesmo caso deve-se clicar no botão AND ou OR para que seja adicionado ao final do campo Rule um espaço em

branco, mais a palavra AND ou OR e outro espaço em branco.

- Quando o usuário clicar no botão "Add task" o sistema verifica se o nome do caso é único no cenário;
- Se o nome for único o sistema grava os dados e atualiza a tela "Add Task";
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;
- Quando o usuário clicar no botão "Cancel" a tela é atualizada;

A tela será a seguinte:

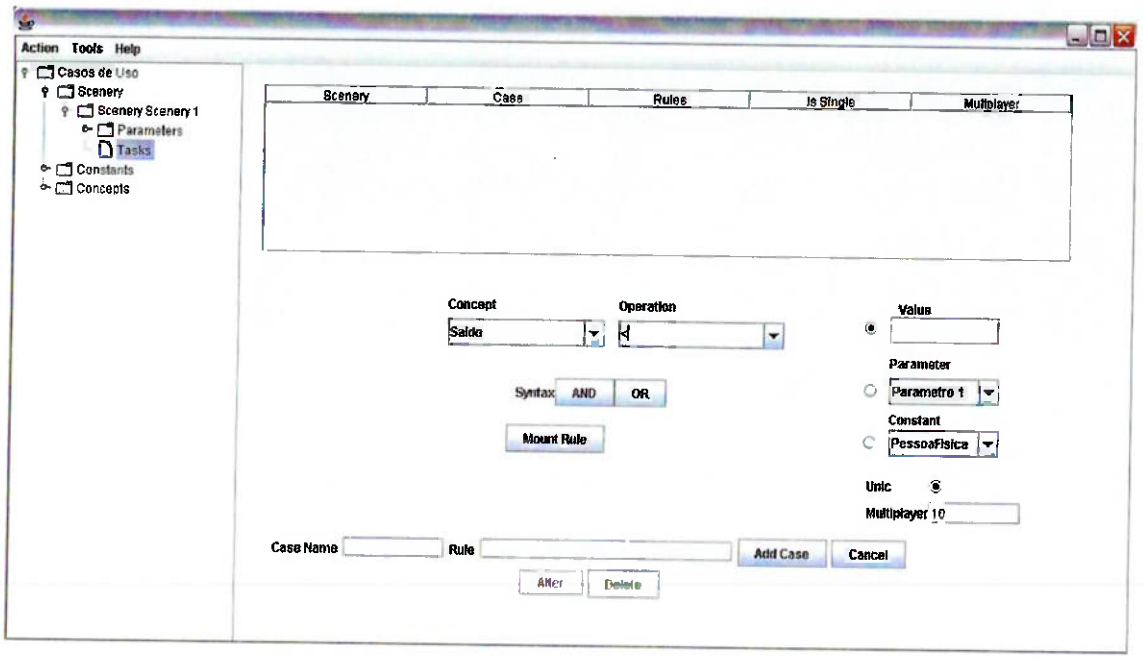


Figura 8.1 Tela de cadastro de Caso(Add Task)

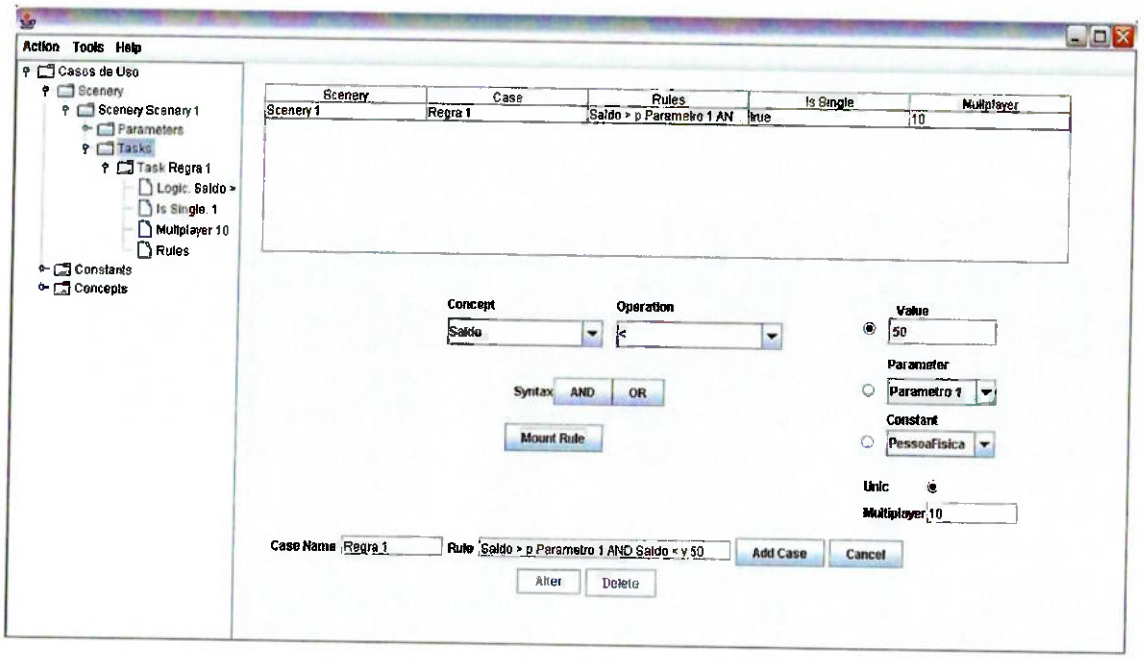


Figura 8.2 Tela após o cadastramento de um Caso

8.4.2 Fluxos Alternativos

A)Exclusão de um Parâmetro

- O ator seleciona um na tabela de Caso
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre o Caso são copiados para os seus respectivos campos de criação;
- Se o ator clicar no botão "Delete" o sistema exibe uma mensagem padrão pedindo a confirmação da exclusão;
- Se o ator confirmar a exclusão a tela "Add Task" é atualizada e os dados são gravados;
- Se o ator não confirmar a exclusão o processo é abortado e nenhuma modificação é feita;

B)Alteração

- O ator seleciona um cenário já cadastrado na tela "Add Task";
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre o Caso são copiados para os seus respectivos campos de criação;
- Quando o usuário clicar no botão "Alter" o sistema verifica se o nome é único no cenário;
- Se for único o sistema atualiza a arvore do Projeto e a tela "Add Task" e grava os dados
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;

8.4.3 Requerimentos Especiais

Não aplicável.

8.4.4 Pós-Condições

Não se aplica.

8.4.5 Pontos de Extensão

Não se aplica.

9 Adicionar, Alterar e Excluir Regra – GMT009

9.1 Breve Descrição

Destina-se a adicionar novas regras aos casos já criados.

9.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

9.3 Pré-Condições

O ator deve ter criado um caso para poder adicionar regras a ele.

9.4 Fluxo de Eventos

Não se aplica.

9.4.1 Fluxo Básico

- O sistema mostra a tela de criação de regras com os botões "Alter" e "Delete" desabilitado e com os campos:
  - ✓ Concept(Check Box com todos os Conceitos cadastrados)
  - ✓ Operation(Pode assumir os valores =, <, >, !=)
  - ✓ Value
  - ✓ Parameter(Check Box com todos os Parâmetros cadastrados naquele Cenário)
  - ✓ Constant(Check Box com todos as Constantes cadastrados no projeto)

- ✓ Unic
- ✓ Multiplayer
- ✓ Case Name
- ✓ Rule
- ✓ Output

- Quando o usuário clicar no botão "Mount Rule" o sistema escreve no campo Rule uma regra conforme os itens selecionados pelo usuário da seguinte forma:
    1. Copia-se para uma String o nome do conceito selecionado com um espaço em branco no final do nome;
    2. Soma-se a String a operação selecionada com um espaço em branco no final;
    3. Verifica-se qual tipo de valor foi selecionado(Value, Constant ou Parameter)
        - Se for Value soma-se a string um v minúsculo com um espaço em branco e soma-se o valor do campo value;
        - Se for Constant soma-se a string um c minúsculo com um espaço em branco e soma-se o nome da constante selecionada;
        - Se for Parameter soma-se a string um p minúsculo com um espaço em branco e soma-se o nome do parâmetro selecionado;
    4. Soma-se a String resultante a String que já contiver no campo Rule
    5. Se quiser adicionar mais uma regra ao mesmo caso deve-se clicar no botão AND ou OR para que seja adicionado ao final do campo Rule um espaço em

branco, mais a palavra AND ou OR e outro espaço em branco.

- Quando o usuário clicar no botão "Add Rule" o sistema verifica se o nome da regra é único no caso;
- Se o nome for único o sistema grava os dados e atualiza a arvore do projeto e a tela "Add Rule";
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;
- Quando o usuário clicar no botão "Cancel" a tela é encerrada;

A tela será a seguinte:

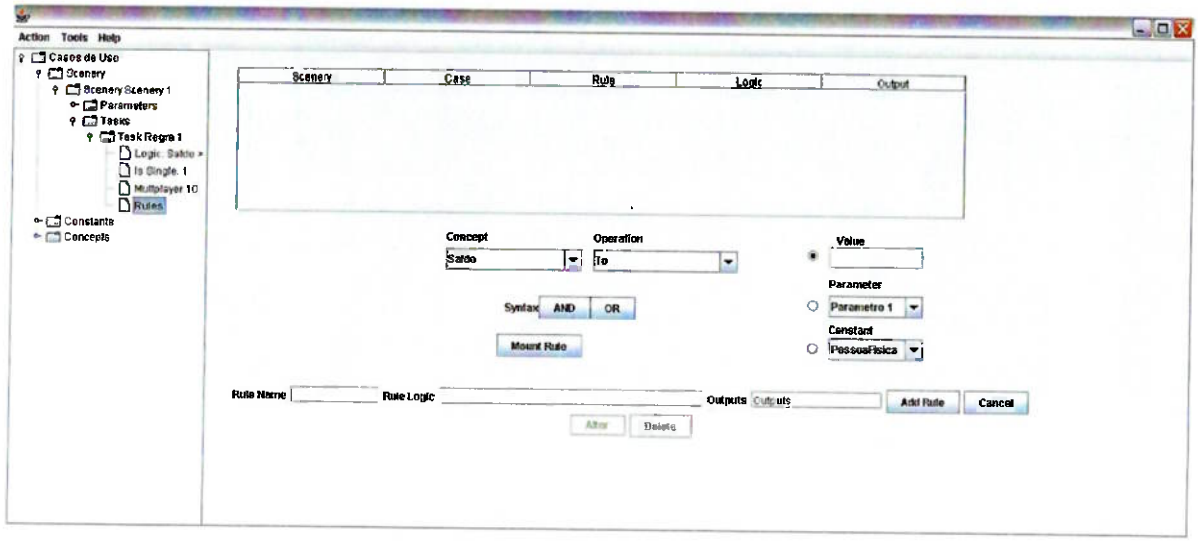


Figura 9.1 Tela de cadastro de Regra(Add Rule)

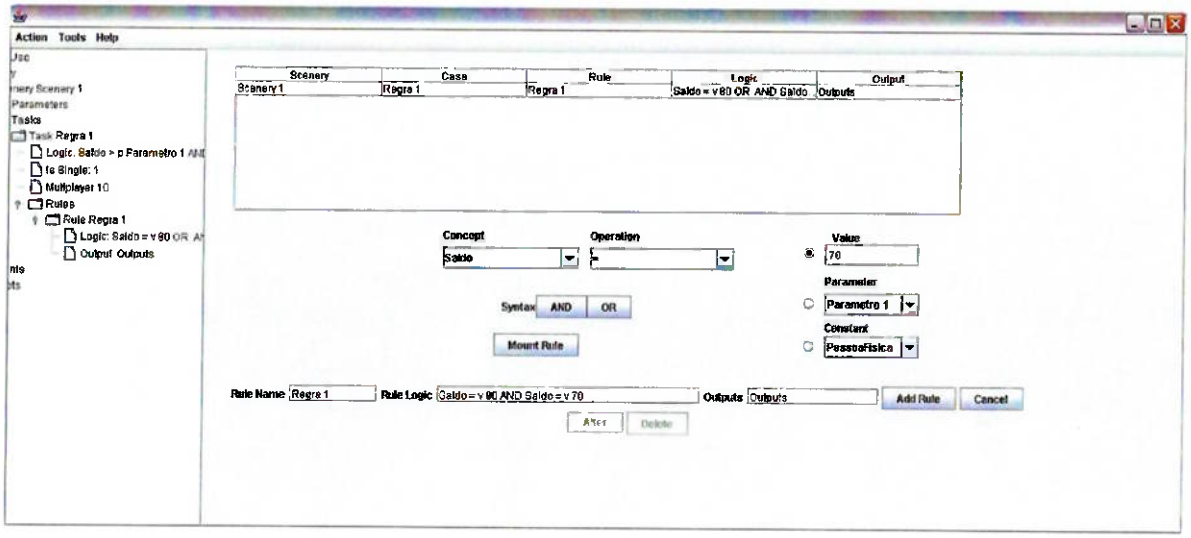


Figura 9.2 Tela após o cadastro de uma regra

9.4.2 Fluxos Alternativos

A)Exclusão de uma Regra

- O ator seleciona uma regra já cadastrada na tabela de Regras
- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre a regra selecionada são copiadas para os seus respectivos campos de criação;
- Se o ator clicar no botão "Delete" o sistema exibe uma mensagem padrão pedindo a confirmação da exclusão;
- Se o ator confirmar a exclusão a tela "Add Rule" é atualizada e os dados são gravados;
- Se o ator não confirmar a exclusão o processo é abortado e nenhuma modificação é feita;

B)Alteração

- O ator seleciona uma regra já cadastrada na tela "Add Rule";

- O sistema habilita os botões "Alter" e "Delete" e todas as informações sobre a regra selecionada são copiadas para os seus respectivos campos de criação;
- Quando o usuário clicar no botão "Accept Alter" o sistema verifica se o nome da regra é único no cenário;
- Se for único o sistema atualiza a tela "Add Rule" e grava os dados
- Se não for único o sistema exibe uma tela pedindo ao usuário para inserir um novo nome;

9.4.3 Requerimentos Especiais

Não aplicável.

9.4.4 Pós-Condições

Não aplicável.

9.4.5 Pontos de Extensão

Nenhum.

10 Gerar Massa de Testes e Roteiro – GMT012

10.1 Breve Descrição

Destina-se a seleção dos cenários escolhidos, do caminho de gravação do arquivo txt e confirmação da geração.

10.2 Atores

Este caso de uso é de uso de todos os usuários do sistema.

10.3 Pré-Condições

Deve existir ao menos uma regra cadastrada.

10.4 Fluxo de Eventos

Não se aplica.

10.4.1 Fluxo Básico

- O sistema mostra a tela "New Generate" com o campo:
  - ✓ Path
- Quando o usuário clica no botão botão "Generate Mass of Test" .o sistema verifica quais cenários estão selecionados e verifica quais registros são relevantes para cada cenário gravando a resposta em um arquivo txt no formato CVS no diretório que está o programa

A tela é a seguinte:

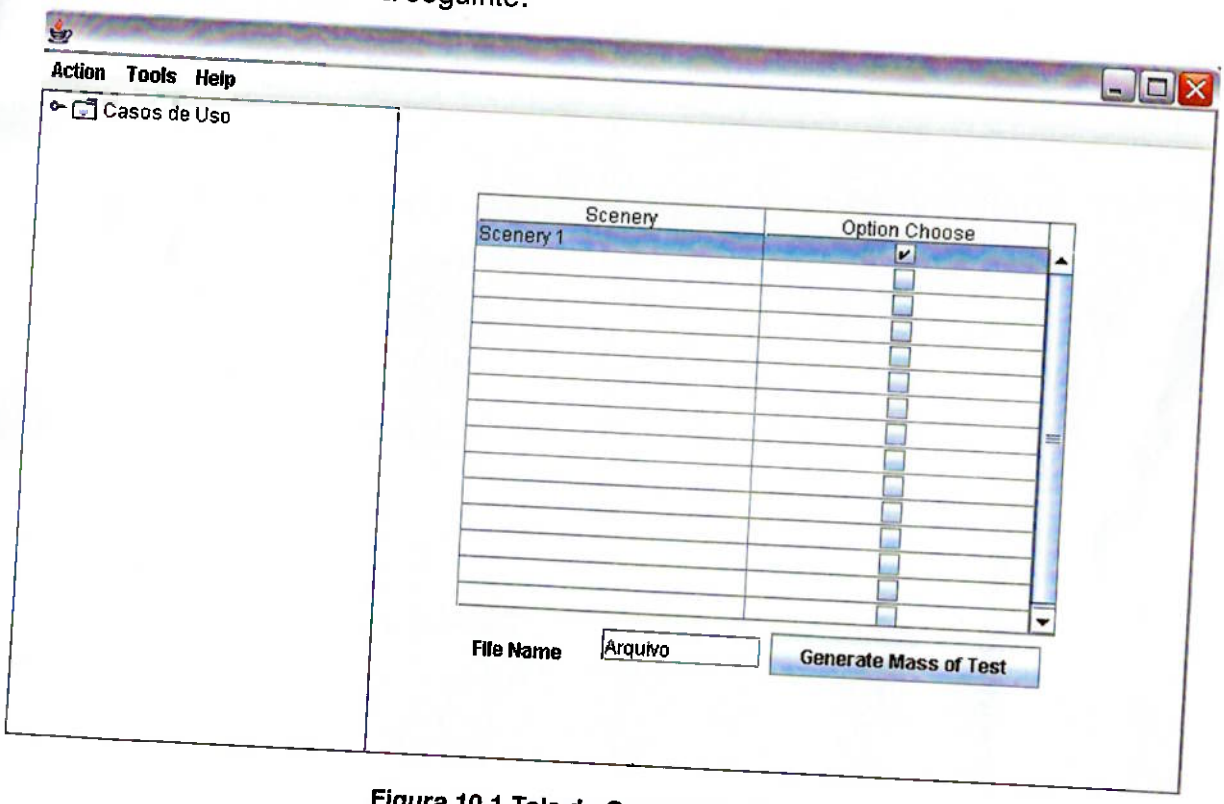


Figura 10.1 Tela de Geraçao da Base

10.4.2 Fluxos Alternativos

Não Aplicável.

10.4.3 Requerimentos Especiais

Este é o core do programa. A primeira parte é composta por funções responsáveis pela leitura e quebra das Strings de regras e casos, gerando informações que comporão a consulta. Assim, a regra é quebrada em tabelas, conceitos, valores operações e operadores lógicos sendo gravada em vetores de Strings.

Em seguida, todas as tabelas são reclassificadas no vetor por ordem alfabética e as funções de montagem são chamadas. Essas funções remontam as regras na forma de chamadas sql. Para tanto conectam-se as regras de casos e suas regras personalizadoras através da variável lógica "and".

Com a chamada montada, procedemos com a consulta e os registros encontrados, advindos das regras cadastradas, são gravados em um arquivo txt, que será disponibilizado para o usuário.

Em resumo, o programa tem duas utilidades. A primeira delas é a de orientação na formulação dos cenários de testes. Procura-se automatizar e padronizar o processo, fazendo com que o usuário pense e estude quais cenários são relevantes à sua pesquisa e qual será a estratégia de testes.

Além disso, através de cadastros simples, montam-se consultas complexas na base de dadso, facilitando e garantindo que a massa de testes gerada realmente é adequada para aquele determinado caso.

10.4.4 Pós-Condições

Deve ser gerado um roteiro de testes e uma base de dados relevantes para este roteiro.

10.4.5 Pontos de Extensão

Nenhum.