UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

A CONSTRUÇÃO DE UM SIMULADOR DE PROCESSOS CONCORRENTES UTILIZANDO DEVS COMO ESTRUTURA FORMAL DE MODELAGEM

RENATO CORREA BRAZ

BLUMENAU
2006

2006/1-17

RENATO CORRÊA BRAZ

A CONSTRUÇÃO DE UM SIMULADOR DE PROCESSOS CONCORRENTES UTILIZANDO DEVS COMO ESTRUTURA FORMAL DE MODELAGEM

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Sistemas de Informação — Bacharelado.

Prof. Mauro Marcelo Mattos, Dr. - Orientador

BLUMENAU
2006

2006/1-17

A CONSTRUÇÃO DE UM SIMULADOR DE PROCESSOS CONCORRENTES UTILIZANDO DEVS COMO ESTRUTURA FORMAL DE MODELAGEM

Por

RENATO CORREA BRAZ

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Mauro Marcelo Mattos, Dr. – Orientador, FURB
Membro:	Prof. Antonio Carlos Tavares – FURB
Membro:	Prof. Miguel Alexandre Wisintainer – FURB

Blumenau, 3 de Julho de 2006

Dedico este trabalho à todos aqueles que me apoiaram nesta jornada.

AGRADECIMENTOS

À minha família, que sempre me apoiou, mesmo longe sempre esteve presente.

Ao Arnaldo Braun, por seus preciosos ensinamentos em Delphi .

Ao meu orientador, professor Mauro Mattos, pela dedicação e por ter acreditado na conclusão deste trabalho.

A todos os amigos, pelo incentivo ao longo desta caminhada.

À minha esposa, Marineusa.

E ao Daniel, meu filho, por transformar a vida numa alegria permanente.

A persistência é o caminho do êxito.

Charles Chaplin

RESUMO

Este trabalho apresenta a fundamentação teórica e descreve a implementação de um *framework* de simulação baseado no formalismo DEVS. O modelo é validado através da simulação de um ambiente multitarefa de processos concorrentes.

Palavras chaves: DEVS. Simulação. Eventos discretos.

ABSTRACT

This work presents the fundamental concepts behind DEVS formalism and describes a framework developed. The framework is validated with a concurrent process model.

Key-words: DEVS. Simulation. Discrete Event.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

DEVS – Discrete Event System Specification

tL – time Last

tN – time Next

CPU – Central Processing Unit

ta – Time Advance

S – Estado

e – Elapsed Time

t – Current Time of Simulation

E/S – Entrada/Saída

SUMÁRIO

1 INTRODUÇÃO

"O sistema operacional é uma camada de software colocada entre o hardware e os programas que executam tarefas para os usuários" (OLIVEIRA; CARISSIMI; TOSCANI, 2000, p. 1). Os programas de sistema, algumas vezes chamados de utilitários, são programas normais executados como processos fora do núcleo do sistema operacional. Segundo Oliveira, Carissimi e Toscani (2000, p. 14), na maioria das vezes, um processo é definido como um programa em execução.

"O correto entendimento dos mecanismos presentes nos sistemas operacionais permite ao profissional de informática uma melhor compreensão de seu ambiente de trabalho, resultando no desenvolvimento de soluções com maior qualidade e eficiência" (SANTOS, 2005, p. 12). Um sistema operacional é demasiado complexo para ser examinado com métodos e ferramentas analíticas tradicionais. As técnicas de simulação e modelagem, por demonstrarem um certo estágio de êxito, podem ser aplicadas como ferramentas de análise.

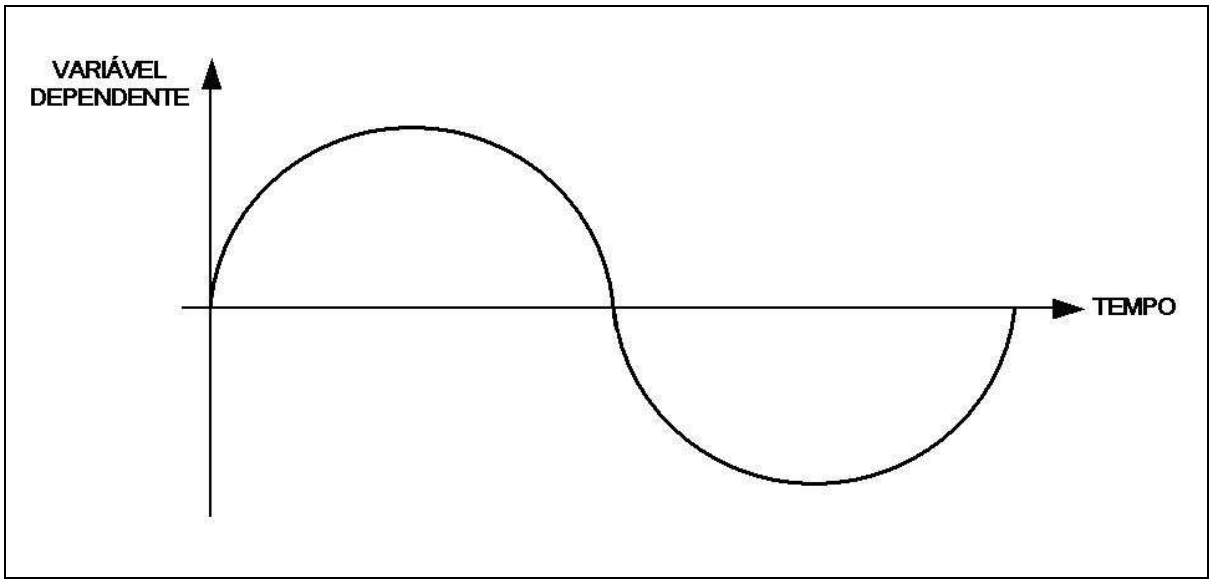
Segundo Prado (1999), a "simulação é a técnica de solução de um problema pela análise de um modelo que descreve o comportamento do sistema usando um computador digital". A simulação pode ser aplicada na situação em que o modelo a ser estudado não possa ser modificado para ser examinado. A simulação permite estudar problemas específicos usando a experimentação virtual.

Segundo Giozza et al (1986, p. 280), a modelagem de um sistema pode ser entendida como sendo uma maneira de simplificar suas características, descartando tudo aquilo considerado irrelevante no sistema.

Segundo Soares (1992, p. 11), os "modelos de um sistema podem ser classificados como modelos de mudança discreta e mudança contínua". Conforme Freitas Filho (2005, p. 18), "estes conceitos estão associados à idéia de sistemas que sofrem mudanças de forma discreta ou contínua ao longo do tempo. A caracterização de um modelo é dada em função da maneira com que ocorrem as mudanças nas variáveis de estado do sistema".

"Em um modelo para simulação contínua, o estado do sistema é representado por variáveis dependentes que mudam continuamente no tempo" (SOARES, 1992, p. 19). Segundo Freitas Filho (2005, p. 21), "nestes modelos, as variáveis de estado podem mudar continuamente ao longo do tempo".

A figura 1 mostra uma variável dependente em um modelo contínuo, com tempo contínuo.

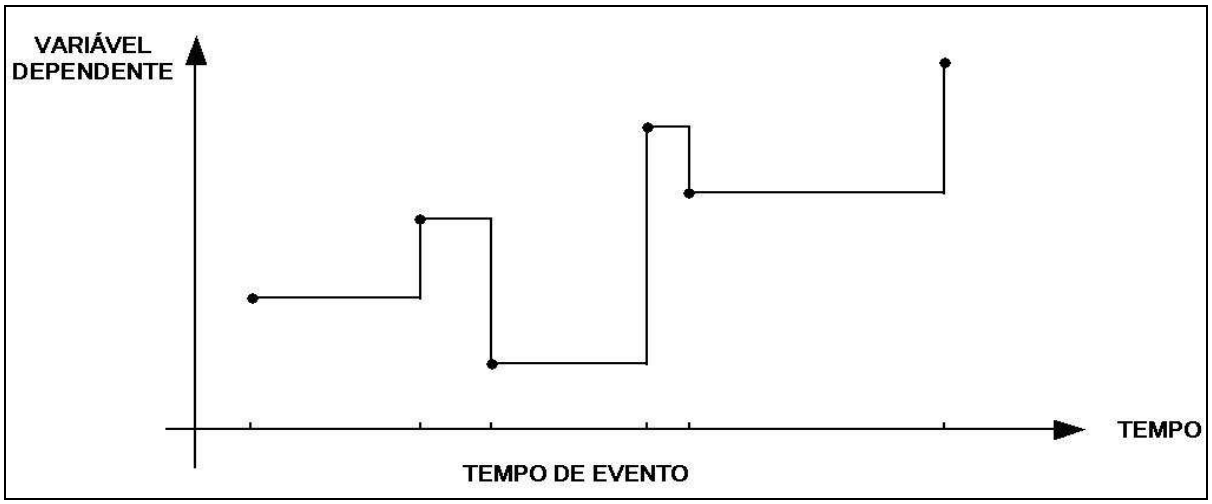


Fonte: Soares (1992, p. 13)

Figura 1 - Variável dependente em um modelo contínuo com tempo contínuo

"Em simulação discreta, o estado do sistema só pode mudar nos tempos de eventos" (SOARES, 1992, p. 15). Segundo Freitas Filho (2005, p. 19), "nestes modelos, as variáveis de estado mantém-se inalteradas ao longo de intervalos de tempo e mudam seus valores somente em momentos bem definidos".

A figura 2 mostra uma variável dependente em um modelo discreto.



Fonte: Soares (1992, p. 12)

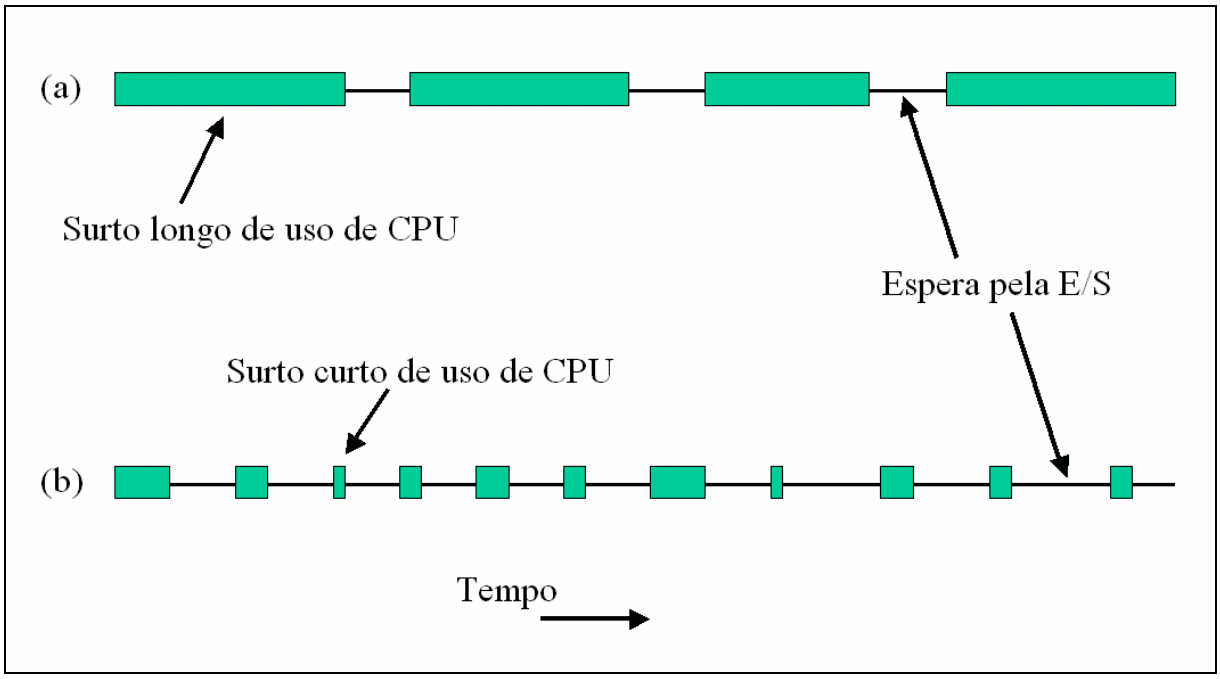
Figura 2 – Variável dependente em um modelo discreto

Segundo Zeigler e Sarjoughian (2002, p. 1, tradução nossa), “a estrutura de um modelo pode ser expressa em uma linguagem matemática chamada formalismo”. Nesse contexto, o formalismo Discrete Event System Specification (DEVS) provê uma forma de especificar um objeto matemático chamado sistema.

O ensino de programação concorrente é uma atividade que demanda uma certa capacidade de abstração por parte dos alunos. Nem sempre é fácil para o aluno, num primeiro momento, compreender as nuances apresentadas pelos problemas clássicos de sincronização de processos descritos na literatura. Desse contexto provém a necessidade de uma ferramenta didática através da qual o aluno possa interpretar o funcionamento dos processos concorrentes.

Tanenbaum (2003, p. 98) caracteriza esta situação através do seguinte exemplo.

> Quase todos os processos alternam surtos de computação com requisições de E/S (de disco), conforme mostra a Figura 3. Em geral, a CPU executa indefinidamente e então é feita uma chamada ao sistema para ler de um arquivo ou escrever nele. Quando a chamada ao sistema termina, a CPU computa novamente até que ela requisite ou tenha de escrever mais dados e assim continua. Perceba que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo a fim de atualizar a tela, ela está computando não fazendo E/S, pois a CPU se encontra em uso. E/S, nesse sentido, é o que ocorre quando um processo entra no estado bloqueado esperando que um dispositivo externo termine o que está fazendo.
>
> O que é importante observar na Figura 3 é que alguns processos, como os da Figura 3(a), gastam a maior parte do tempo computando, enquanto outros, como os da Figura 3(b), passam a maior parte de seu tempo esperando E/S. Os primeiros são chamados **orientados à CPU**; os últimos são os **orientados à E/S**. Os processos orientados a CPU apresentam, em geral, longos surtos de uso da CPU e esporádicas esperas por E/S, já os processos orientados à E/S têm pequenos surtos de uso da CPU e esperas freqüentes por E/S. Note que o fator principal é o tamanho do surto de CPU, não o tamanho do surto de E/S. OS processos orientados à E/S, são assim chamados porque, entre uma requisição e outra por E/S, eles não realizam muita computação, não porque tenham requisições por E/S especialmente demoradas. O tempo para a leitura de um bloco de disco é sempre o mesmo, independentemente do quanto demore processar os dados que chegam depois.
>
> Convém observar que, à medida que as CPUs se tornam mais rápidas, os processos tendem a ficar mais orientados à E/S. Esse efeito ocorre porque as CPUs estão ficando muito mais rápidas que os discos. Como conseqüência, o escalonamento de processos orientados à E/S deverá ser um assunto mais importante no futuro. A idéia básica é que, se um processo orientado à E/S quiser executar, deve ser rapidamente dada a ele essa oportunidade, pois assim ele executará suas requisições de disco, mantendo o disco ocupado. (TANENBAUM, 2003, p. 98, grifo do autor).



Fonte: Tanenbaum (2003, p. 98)

Figura 3 - Surtos de uso da CPU alternam-se com períodos de espera por E/S. (a) Um processo orientado à CPU. (b) Um processo orientado à E/S.

Tanenbaum (2003,p.99) destaca que um outro problema importante está relacionado à questão de quando escalonar um processo. Um tópico fundamental, relacionado ao escalonamento, é o momento certo de tomar as decisões de escalonar. Claro que há uma variedade de situações nas quais o escalonamento é necessário. Primeiramente, quando se cria um novo processo, é necessário tomar uma decisão entre executar o processo pai ou o processo filho. Como ambos os processos estão no estado pronto, essa é uma decisão normal de escalonamento e pode levar a escolha de um ou de outro – isto é, o escalonador pode escolher legitimamente executar o pai ou o filho (Tanenbaum, 2003, p.99).

Neste sentido, o presente projeto tem por objetivo desenvolver um simulador didático de processos concorrentes utilizando o formalismo DEVS como infra-estrutura (*framework*) de modelagem. A ferramenta de simulação permite ao aluno alterar os parâmetros de configuração (número de processos, tempo de acesso a periféricos, perfil de processo (IO/bound e CPU/bound), entre outros) e analisar o comportamento do sistema sob as condições configuradas. Esta análise permitirá ao aluno, por exemplo, avaliar o tempo de resposta de determinadas aplicações em função da carga do sistema, considerada a configuração aplicada ao modelo.

1.1 OBJETIVOS

O objetivo geral deste trabalho é construir uma *engine* de simulação que permita a construção de um simulador didático de processos concorrentes utilizando DEVS como estrutura formal de modelagem.

Os objetivos específicos do trabalho são:

a) implementar a *engine* de simulação baseada no formalismo DEVS;

b) construir e implementar um modelo de simulação de processos concorrentes para validar a *engine* DEVS;

c) permitir a parametrização de variáveis de controle vinculadas ao modelo atômico.

1.2 RELEVÂNCIA DO TRABALHO

O aspecto de maior relevância neste projeto refere-se ao fato da utilização de um modelo formal de especificação de simuladores. Em um momento em que a área de engenharia de software reconhece a necessidade do emprego de métodos formais no processo de desenvolvimento de software, a iniciativa do emprego do modelo DEVS é plenamente justificada.

Sob o aspecto didático, a construção de um simulador de processos concorrentes possibilitará ao aluno experimentar várias situações relativas aos mecanismos de gerenciamento de processos (escalonamento, *deadlock*, sincronização) e desse modo complementar às explicações teóricas de sala de aula, da disciplina de sistemas operacionais (S.O.).

1.3 ESTRUTURA DO TRABALHO

Este primeiro capítulo de introdução apresentou uma contextualização do trabalho, destacando e apresentando o assunto correspondente bem como os objetivos almejados.

O capítulo 2 discorre sobre o embasamento para o modelo construído.

O capítulo 3 apresenta a especificação do sistema construído e o capítulo 4 apresenta um estudo de caso descrevendo o funcionamento da ferramenta bem como as considerações finais

O capítulo 5 apresenta a conclusão, as limitações e futuras extensões para este trabalho.

2 REVISÃO BIBLIOGRÁFICA

A seguir são apresentados os principais aspectos relacionados a este projeto objetivando estabelecer um domínio conceitual necessário ao entendimento do mesmo.

2.1 SIMULAÇÃO

Segundo Mello (2001, p.11) a simulação de sistemas tem como objetivo geral permitir a execução de experimentos sobre modelos que representam sistemas reais. Um modelo pode ser interpretado como uma descrição de um sistema. A justificativa para a construção e realização de experimentos sobre modelos é baseada na existência de problemas de ordem prática que impedem, oneram, ou dificultam a interferência em sistemas reais, dependendo de suas características. Por exemplo, quando um sistema ainda está na fase de projeto, quando a interferência em um sistema pode resultar em custos altos sem garantias de melhoras, ou, também, quando o sistema real apresenta alto risco. Deste modo, o uso de estratégias de simulação permite que informações com bons níveis de confiabilidade sejam geradas, ademais, permite que testes/experimentos em busca de soluções otimizadas sejam executadas sobre modelos que podem ser mantidos sob bons níveis de controle.

Nesse contexto, as soluções de simulação por computador vêm adicionar ganhos elevados na medida em que, à luz do potencial de processamento, modelos de representação com alta fidelidade, ou alto grau de detalhamento, podem ser alcançados e tratados com sucesso. Em sistemas computacionais, os modelos de simulação podem ser executados em ambientes centralizados (simulação centralizada) ou em ambientes distribuídos (simulação distribuída). Em ambos os ambientes de simulação um modelo de representação pode ser dividido em sub-modelos que executam em diferentes computadores (Mello,2001,p.11).

Segundo Prado (1999, p. 96), uma das justificativas para o uso da simulação está na “inviabilidade da interferência com o sistema real. Trata-se daquela situação em que tentar alterar o sistema existente, sem ter uma certeza de que a alteração vai dar certo, pode significar um alto risco de prejuízo”.

Para Freitas Filho (2005, p. 5), a simulação é uma maneira de “prever o comportamento futuro dos sistemas usando modelos, isto é, antecipar os efeitos produzidos por alterações [...]”. Dessa forma pode-se aplicar modificações no modelo e obter resultados através de experimentações virtuais. Conforme Giozza et al (1986, p. 280), a simulação avalia

o desempenho do modelo quando “isto pode não ser possível se o sistema ainda não estiver implementado ou se não puder ser perturbado”. A experimentação permite que o modelo seja avaliado tantas vezes quanto necessário, sem a necessidade de se aplicar experimentos com o próprio sistema.

Soares (1992, p. 2) define o sistema como sendo “uma coleção de itens, entre os quais se possa encontrar ou definir alguma relação, que são objeto de estudo ou interesse”. Prado (1999, p. 94) emprega o termo modelo para “significar a representação de um sistema”.

Segundo Soares (1992, p. 3), os “modelos para simulação podem ser definidos como aqueles representados por uma estrutura matemática/lógica, que pode ser exercitada de forma a mimetizar o comportamento do sistema”.

2.2 CONCEITOS DE MODELAGEM

Conforme Mello (2001, p.12), “o primeiro conceito de modelagem diz respeito à própria composição de um modelo de simulação. Da mesma forma que um sistema, um modelo é composto de um conjunto de entidades que interagem entre si. As entidades de um sistema são representações do sistema real, e o conjunto de entidades de um modelo pode representar todo ou apenas uma parte de um sistema real, desde que seja suficientemente abrangente para alcançar os objetivos de um estudo em particular sobre o sistema”.

As propriedades de uma entidade são descritas através de seus atributos, e o comportamento de uma entidade é definido de acordo com o modo de como seu estado se altera ao longo do tempo em função da ocorrência de eventos (tempo de evento). Ou seja, as mudanças de estado de uma entidade (e consequentemente do modelo) são dependentes da ocorrência de eventos. É através da especificação dos eventos que um projetista modela o comportamento de um modelo de simulação.

Os eventos podem ocorrer de modo discreto, contínuo ou híbrido. No modo discreto, os eventos ocorrem apenas em determinados instantes (não contínuos) entre intervalos irregulares de tempo. No modo contínuo os eventos ocorrem em todos os instantes de tempo. E o modo híbrido é uma combinação dos dois primeiros modos num mesmo modelo.

Um modelo pode ser dinâmico ou estático. Modelos dinâmicos podem ser ditos dependentes do tempo (variam no tempo). O tempo de simulação é o elemento que determina a seqüência de eventos de uma simulação. A variável que possui o valor do tempo simulado é denominada relógio de simulação. Modelos estáticos não utilizam a variável de tempo.

Modelos também podem ser estocásticos ou determinísticos. Modelos estocásticos fazem uso de probabilidade para determinar o valor de alguns ou todos os atributos das entidades e/ou do tempo. Por isso, diferentes exercícios de um mesmo modelo podem gerar resultados diferenciados. Modelos determinísticos não utilizam funções de probabilidade, com efeito, diferentes exercícios de um mesmo modelo determinístico geram resultados iguais.

Quanto à descrição do comportamento das entidades, um projetista o descreve definindo uma seqüência de reações em função da ocorrência de eventos. Para esta descrição, podem ser utilizadas algumas estratégias (visões) baseadas na programação de eventos, na varredura de atividades, e na iteração entre processos.

Na visão baseada na programação de eventos, são descritos os eventos que podem ocorrer e a relação de causa e efeito entre eles. Novos eventos podem ser gerados em tempo de execução do modelo, ou o algoritmo de simulação pode programar um conjunto dos eventos, armazenando-os em uma lista ordenada pelo tempo. Então o algoritmo de simulação seleciona eventos (um de cada vez) da lista, atualiza o relógio de simulação, e ativa o procedimento correspondente ao evento.

Um estudo baseado em simulação não pode ser considerado como um processo seqüencial simples. Principalmente porque um estudo deste tipo, além dos resultados de simulação, oferece uma oportunidade de entender mais profundamente um sistema em estudo. Esta característica geralmente provoca retrocessos no estudo de simulação em situações onde são detectadas falhas em uma fase anterior, obrigando a regressão parcial do estudo e rever possíveis reformulações para a solução de problemas.

2.3 DISCRETE EVENT SYSTEM SPECIFICATION (DEVS)

O formalismo Discrete Event System Specification (DEVS) provê uma forma de especificar um objeto matemático chamado sistema. Basicamente um sistema possui uma base de tempo, entradas, estados, saídas e funções para determinar os próximos estados e saídas uma vez fornecidos os estados e entradas atuais. DEVS é um *framework* formal de modelagem e simulação. É baseado em conceitos de sistemas dinâmicos genéricos que podem integrar uma infra-estrutura de execução paralela e distribuída, propostos por Zeigler e Sarjoughian.

Segundo Zeigler e Sarjoughian (2002, p. 1, tradução nossa), “a principal característica apresentada pelo formalismo DEVS está na facilidade para a especificação dos parâmetros do modelo de simulação a eventos discretos”.

Os itens básicos de dados produzidos por um sistema ou modelo no contexto **DEVS** são: segmentos de tempo. Estes segmentos de tempo constituem-se em mapeamentos de intervalos de tempo (definidos sobre uma base de tempo especificada) a valores de uma ou mais variáveis. Estas variáveis podem ser ou observadas ou medidas (calculadas).

Dois aspectos importantes relacionados ao tempo e ao formalismo são destacados por Ziegler e Sarjoughian (2002) na seguinte afirmação:

> *"A estrutura de um modelo pode ser expressa em uma linguagem matemática denominada formalismo. O formalismo de eventos discretos concentra-se nas alterações do valor de variáveis e gera segmentos de tempo constantes . Assim, um evento é uma alteração no valor de uma variável que ocorre instantaneamente. Em essência, o formalismo DEVS define como gerar novos valores para variáveis e os tempos em que estes valores devem ser percebidos. Um aspecto importante do formalismo é que os intervalos de tempo entre as ocorrências dos eventos são variáveis contrastando com sistemas a tempo discreto onde o incremento de tempo é geralmente é definido por uma constante"*

No formalismo **DEVS**, devem ser especificados (ZIEGLER e SARJOUGHIAN**,**2002):

- os modelos básicos a partir dos quais modelos maiores podem ser construídos e;
- como estes modelos são conectados de modo a formar uma hierarquia.

Segundo Ziegler e Sarjoughian (2002), nas linguagens de simulação tradicionais, deve-se conceber uma visão do modelo como possuindo portas de entrada e saída através das quais toda a interação com o ambiente é mediada. Em **DEVS**, são os eventos que determinam os valores que devem aparecer nas portas de saída. Mais especificamente, quando eventos externos (chegando de fora do modelo) são recebidos em suas portas de entrada, a descrição do modelo deve determinar como responder as mesmas. Além disso, eventos internos ocorrendo dentro do modelo, além de mudar o estado do modelo, manifestam-se como eventos nas portas de saída de modo que os efeitos podem ser transmitidos para outros componentes do modelo.

Um modelo básico **DEVS** pode ser decomposto em sub-modelos atômicos, os quais são combinados através de modelos compostos (*coupled*). Um modelo básico DEVS é descrito formalmente da seguinte forma:

$$\mathbf{M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda\ , ta \rangle}$$

onde:

- **X:** representa o conjunto das portas de entrada (ou valores de entrada) através das quais os eventos externos são recebidos;

- **S:** representa o conjunto de estados (variáveis e parâmetros); duas variáveis estão geralmente presentes, *phase* e *sigma*, as quais, na ausência de eventos externos permitem ao sistema permanecer na *phase* atual por uma duração de tempo determinada por *sigma*;
- **Y:** representa o conjunto de portas de saída (ou valores de saída) através dos quais os eventos externos são sentidos;
- δ_{int}**:** é a função de transição interna, a qual especifica qual será o próximo estado para o qual o controle será transferido após o prazo definido na função de avanço de tempo ter esgotado; formalmente esta afirmação pode ser representada da seguinte forma:

  $\delta_{int}: S \rightarrow S$
- δ_{ext}: é a função de transição externa a qual especifica como o sistema muda de estado quando uma entrada é recebida – o efeito é colocar o sistema em uma nova *phase* e *sigma*, escalonando-o para uma próxima transição interna. Neste caso o próximo estado é calculado a partir do estado atual, da porta de entrada, do valor do evento externo e do tempo decorrido no estado atual. Isto pode ser colocado formalmente da seguinte forma:

  $Q \times X^b \rightarrow S$ onde

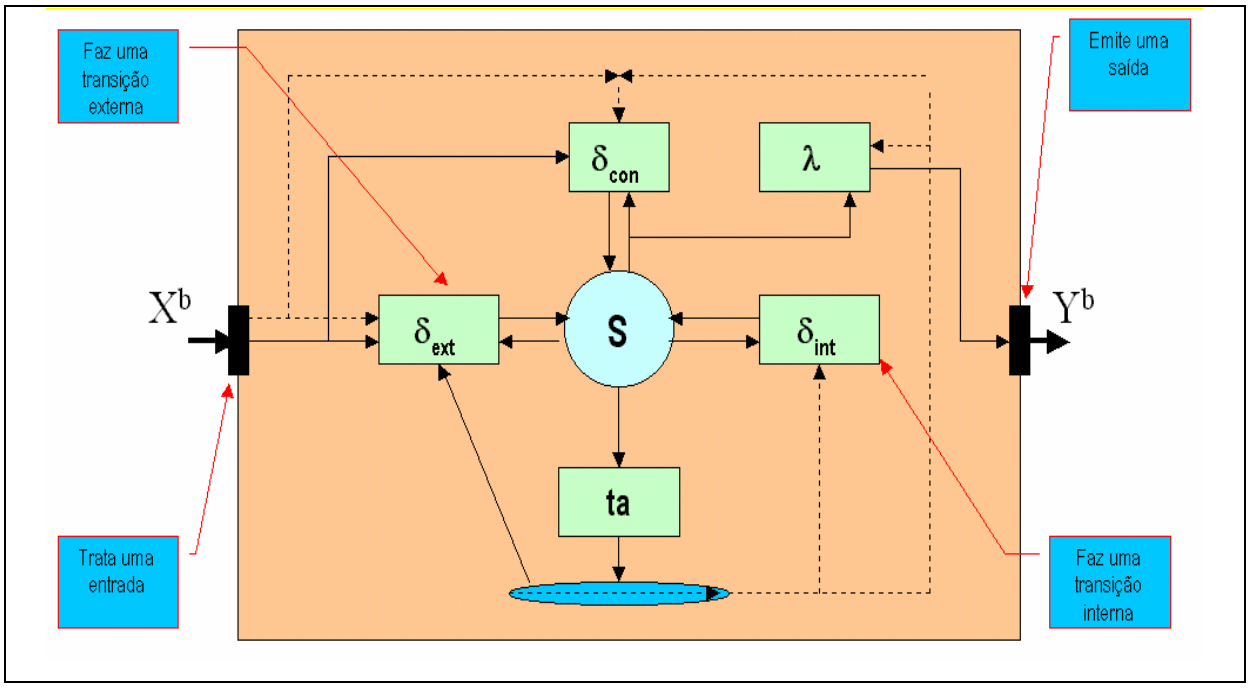
  Q = $\{(s,\boldsymbol{e}) \mid s \in S, 0 \leq \boldsymbol{e} \leq ta(s)\}$ é o conjunto de estados;

  e representa o tempo decorrido (*elapsed time*) desde a última transição

  $\mathbf{X}^b$ denota a coleção de conjuntos (*bags*) sobre X, nos quais alguns elementos podem ocorrer mais que uma vez.
- **λ:** é a função de saída a qual gera uma saída externa antes da transição interna efetivar-se; formalmente esta função é definida como:

  $\lambda: S \rightarrow Y^b$**.**
- ta: é a função que controla o *timing* das transições internas; quando a variável de estado sigma está presente, esta função retorna o valor de sigma; a especificação formal é:

  $ta: S \rightarrow R^+_{0,\infty}$



Fonte: Adaptado de Ziegler e Sarjoughian (2002)

Figura 4 – Estrutura de um modelo DEVS.

A interpretação destes elementos, ilustrada na Figura 4, é a seguinte:

- a qualquer tempo o sistema está em algum estado s;
- se não houverem eventos externos, o sistema permanecerá no estado s por ta(s) tempo; observe-se que ta(s) pode ser um número real, mas também pode receber valores 0 e ∞. No primeiro caso, diz-se que o modelo está em um estado transitório; no segundo caso, o sistema permanecerá no estado s para sempre até que algum evento externo o interrompa – neste caso o modelo está num estado passivo.
- quando o tempo de "descanso" (*resting*) termina, ou seja, quando o tempo decorrido (*elapsed time*) e = ta(s), o sistema apresenta as saídas nas portas de saída através da função $\lambda(s)$ e altera o estado através da função δ_{int} (as saídas são disponibilizadas imediatamente antes da transição);
- se um evento externo $x \in X^b$ ocorrer antes da expiração do tempo, ou seja, quando o sistema está em um estado (s,e) com $\boldsymbol{e} \leq ta(s)$, o sistema executa a transição de estado através da função $\delta_{ext}(s,e,x)$.
- isto implica que, a função de transição interna estabelece o novo estado do sistema quando não ocorreram eventos desde a última transição, ao passo que, a função de transição externa estabelece o novo estado do sistema quando um evento externo ocorreu.

Para Barros (2002) como um evento externo $x \in \mathbf{X}^b$ representa o conjunto de elementos de **X**, significa que um ou mais elementos podem ocorrer nas portas de entrada de um modelo ao mesmo tempo o modelo deve ser estendido com uma função extra denominada função de confluência (δ_{con}) a qual é aplicada quando uma entrada é recebida ao mesmo tempo em que uma transição interna está para ocorrer (a situação *default* é aplicar a função de transição interna antes de aplicar a função de transição externa). O modelo formal que suporta esta facilidade é apresentado a seguir:

$$\mathbf{M} = \langle \mathbf{X}, \mathbf{S}, \mathbf{Y}, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, \mathbf{ta} \rangle ,$$

onde

- $\delta_{con} : \mathbf{Q} \times \mathbf{X}^b \rightarrow S$ e

  $\mathbf{Q} = \{(s, \boldsymbol{e}) \mid s \in S, 0 \leq \boldsymbol{e} \leq ta(s) \}$ é o conjunto de estados;

  $\boldsymbol{e}$ representa o tempo decorrido (*elapsed time*) desde a última transição

  $\mathbf{X}^b$ denota a coleção de conjuntos (*bags*) sobre X, nos quais alguns elementos podem ocorrer mais que uma vez.

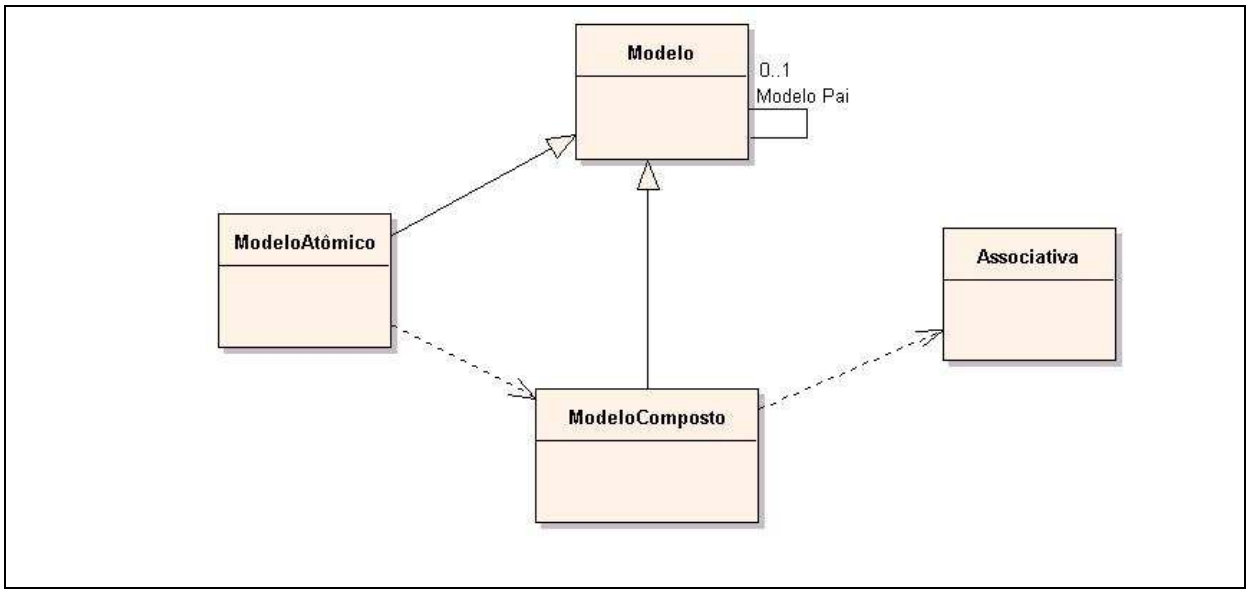
Assim é possível estabelecer comportamentos quando as duas condições a seguir ocorrem simultaneamente:

- A chegada de um evento externo e
- O tempo decorrido do modelo atinge o seu valor máximo.

O *framework* DEVS é constituído por especificações de modelos atômicos, modelos compostos e a associação entre os modelos (*coupling*), conforme demonstrado na figura 4.

Conforme Zeigler e Sarjoughian (2002, p. 7, tradução nossa), “existem dois tipos de componentes: modelos atômicos e modelos compostos. Os modelos atômicos são denominados modelos básicos dentro do formalismo DEVS”.

Um conjunto de modelos básicos ou atômicos combinam-se para formar um modelo composto ou *coupled*.



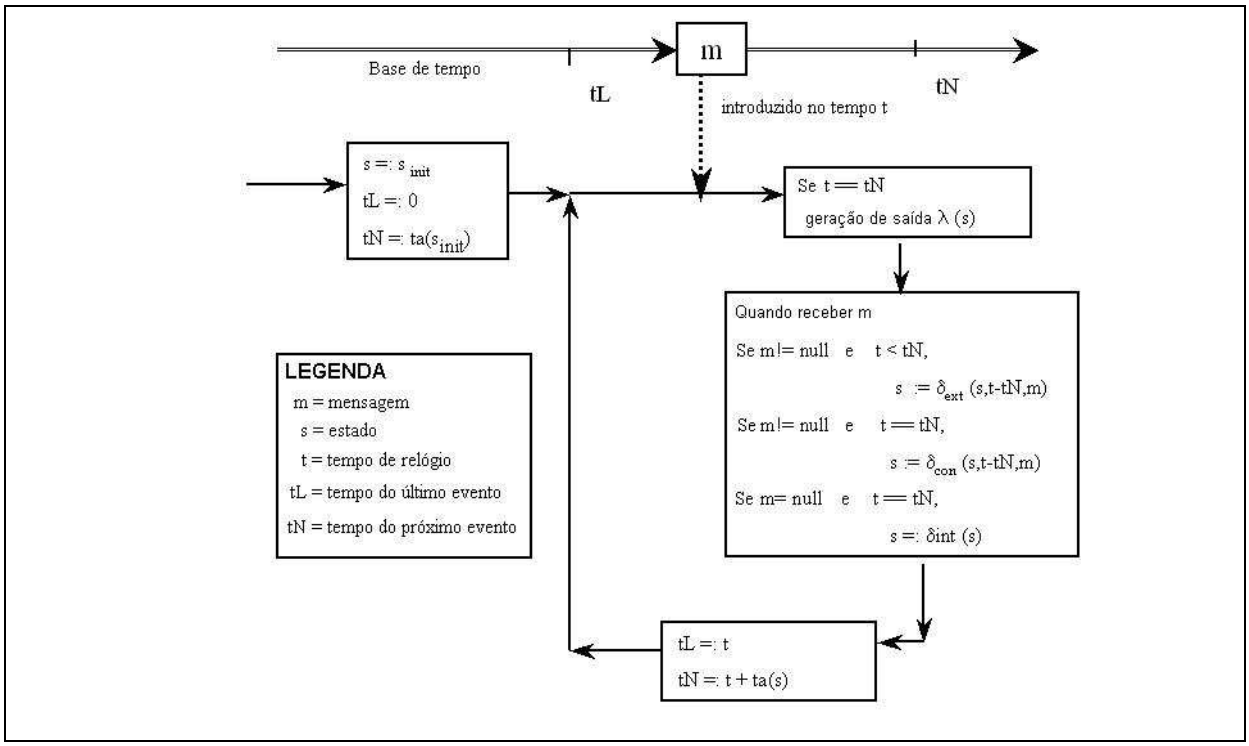
Fonte: Filippi, Chiari e Bisgambilia (2002, p. 3)

Figura 5 - Especificação da estrutura do *framework* DEVS

Um modelo *coupled* especifica como as entradas e as saídas dos componentes se conectam, de modo a formar uma hierarquia, como demonstrado na Figura 5.

2.3.1 FUNCIONAMENTO DO SIMULADOR DO MODELO ATÔMICO

Para cada modelo atômico, existe um controlador denominado simulador o qual é responsável por intermediar toda a comunicação entre o modelo atômico e os demais componentes da hierarquia. A figura 6 demonstra o mecanismo de funcionamento do simulador atômico.



Fonte: Zeigler e Sarjoughian (2003, p. 87)

Figura 6 - Modelo de funcionamento de um simulador atômico

O Quadro 1 apresenta o pseudo-código que descreve o comportamento de um simulador de um modelo atômico conforme especificado em Ziegler (2000, pg.178).

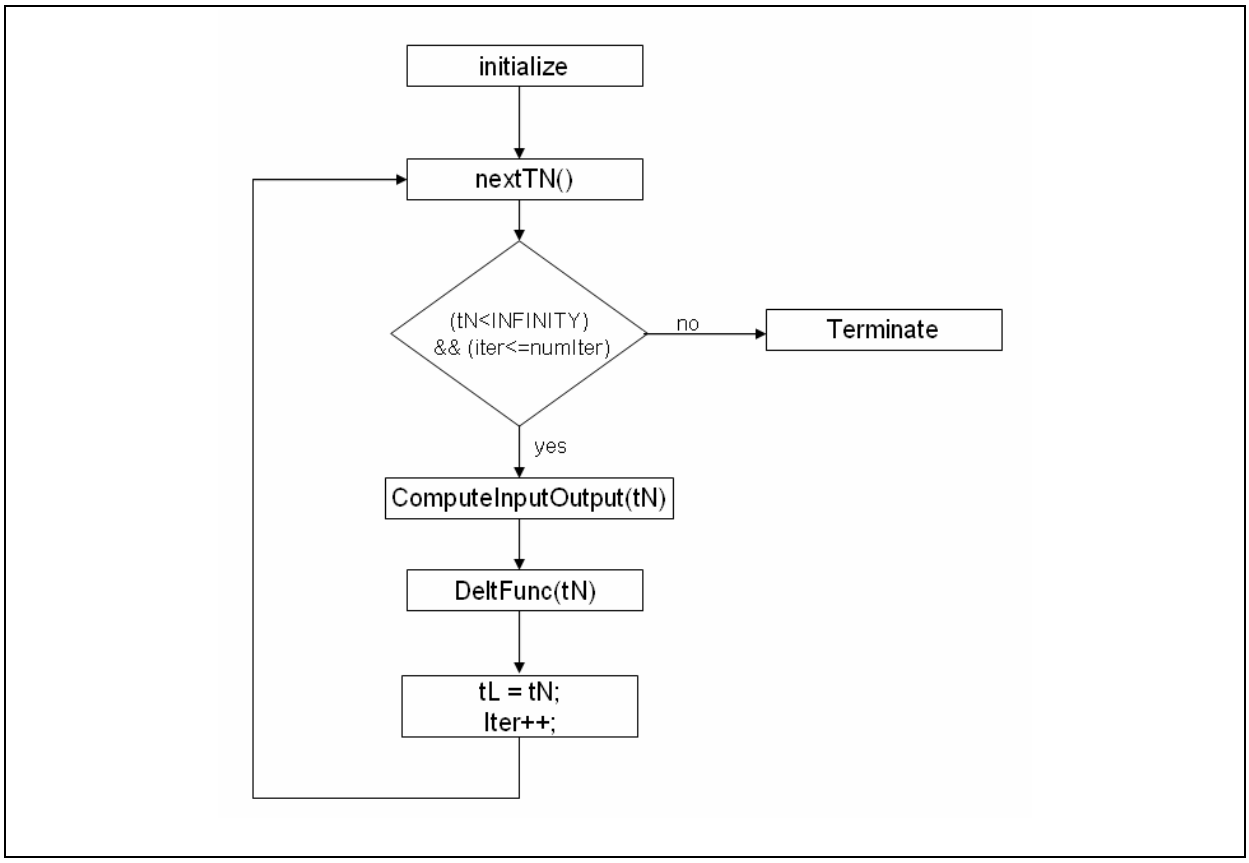
```
Devs-simulator

 Variables:
   parent      // parent coordinator
   tl          // time of last event
   tn          // time of next event
   DEVS        // associated model With total state (s, e)
   y           // current output value of the associated model
When receive i-message (i, t) at time t
  tl = t – e
  tn = tl + ta(s)
When receive *-message (*, t) at time t
  if t != tn then
    error: bad synchronization
  Y =  λ (s)
  Send y-message (y, t) to parent coordinator
  s = δint (s)
  tl = t
  tn = tl + ta(s)
when receive x-message (x,t) at time t with input value x
  if not (tl ≤ t ≤ tn) then
    error: bad synchronization
  e = t – tl
  s = δext(s, e, x)
  tl = t
  tn = tl + ta(s)
end Devs-simulator
```

Quadro 1 – Pseudo-código especificando o comportamento de um simulador de modelos atômicos.

Segundo Hu (2002, p. 2, tradução nossa), “um cenário de simulação simplificado é simular um modelo atômico de modo rápido, nesse caso, um AtomicSimulator é necessário. Na função de construção desse modelo AtomicSimulator, o modelo atômico é associado ao AtomicSimulator. No método initialize(), o AtomicSimulator inicializa o modelo atômico setando tL para 0 e tN para myModel.ta()”.

Ainda conforme Hu (2002, p. 2, tradução nossa),”após a inicialização, como pode-se ver na Figura 7, o AtomicSimulator tem um laço de repetição. Em cada repetição (chamado ciclo de simulação), ele executa o método nextTN() para obter o próximo tN do seu modelo atômico, o método computeInputOutput() para solicitar que o modelo atômico gere sua saída e DeltFunc, para executar o método de transição do modelo”.



Fonte: Hu (2006, p.2)

Figura 7 - Ciclo de simulação do AtomicSimulator.

2.3.2 FUNCIONAMENTO DO SIMULADOR DO MODELO COUPLED

Segundo Hu (2002, p. 2, tradução nossa),”um modelo atômico é um componente básico. Quando da associação de diversos modelos atômicos, obtêm-se um modelo *coupled*. Dessa forma, um modelo *coupled* contêm diversos componentes. Esses componentes podem ser modelos atômicos ou outro modelo *coupled* (construção hierárquica)”.

Para um modelo *coupled* com modelos atômicos, um mecanismo coordenador é associado a ele e, para cada componente básico, é associado um mecanismo simulador. No modelo de protocolo de simulação DEVS, o coordenador é responsável por escalonar os simuladores através do ciclo de atividades de simulação.

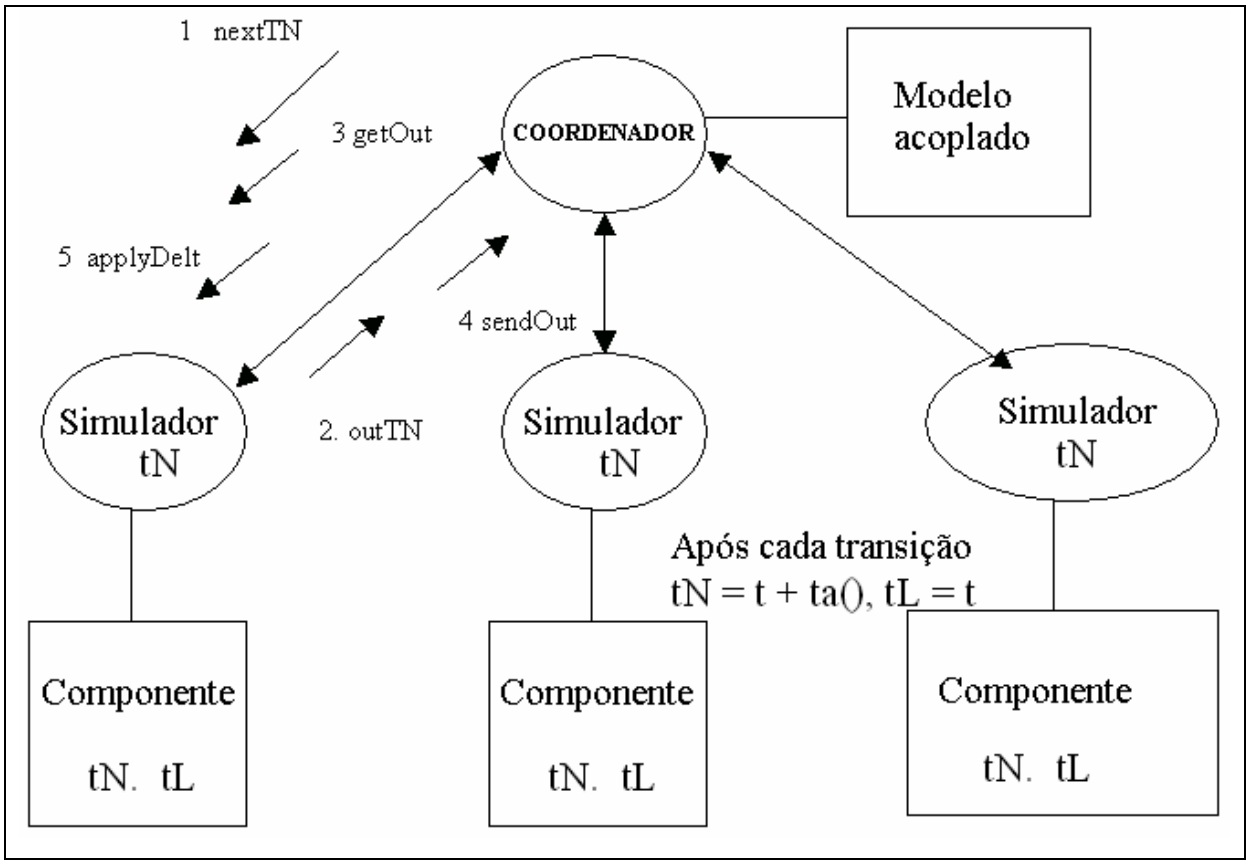
Conforme Zeigler e Sarjoughian (2003, p. 87), o funcionamento do protocolo de simulação é executado através dos passos descritos a seguir.

O mecanismo coordenador solicita o tempo do próximo evento (tN) para cada um dos simuladores associados aos modelos atômicos. Todos os simuladores respondem à esse pedido do coordenador, enviando os seus tempos, indicando a próxima ocorrência de evento.

O coordenador envia a cada mecanismo simulador uma mensagem contendo o tempo global tN (o menor entre os tempos de próximo evento retornados pelos simuladores). Cada simulador verifica se a execução do evento é eminente, ou seja, o tN do mecanismo simulador é igual ao tN global). Se for igual, é retornada a saída do seu modelo associado através de uma mensagem enviada ao mecanismo coordenador.

O coordenador usa as especificações de modelos associados para distribuir as saídas de volta aos simuladores. Para aqueles mecanismos simuladores que não receberam nenhuma entrada, então a mensagem enviada a ele é nula.

A Figura 8 demonstra o mecanismo de funcionamento do protocolo de simulação DEVS.



Fonte: Zeigler e Sarjoughian (2003, p. 87)

Figura 8 - Protocolo de simulação DEVS

O Quadro 1 apresenta o pseudo-código descrevendo o comportamento do coordenador conforme especificado em Ziegler (2000, p.180).

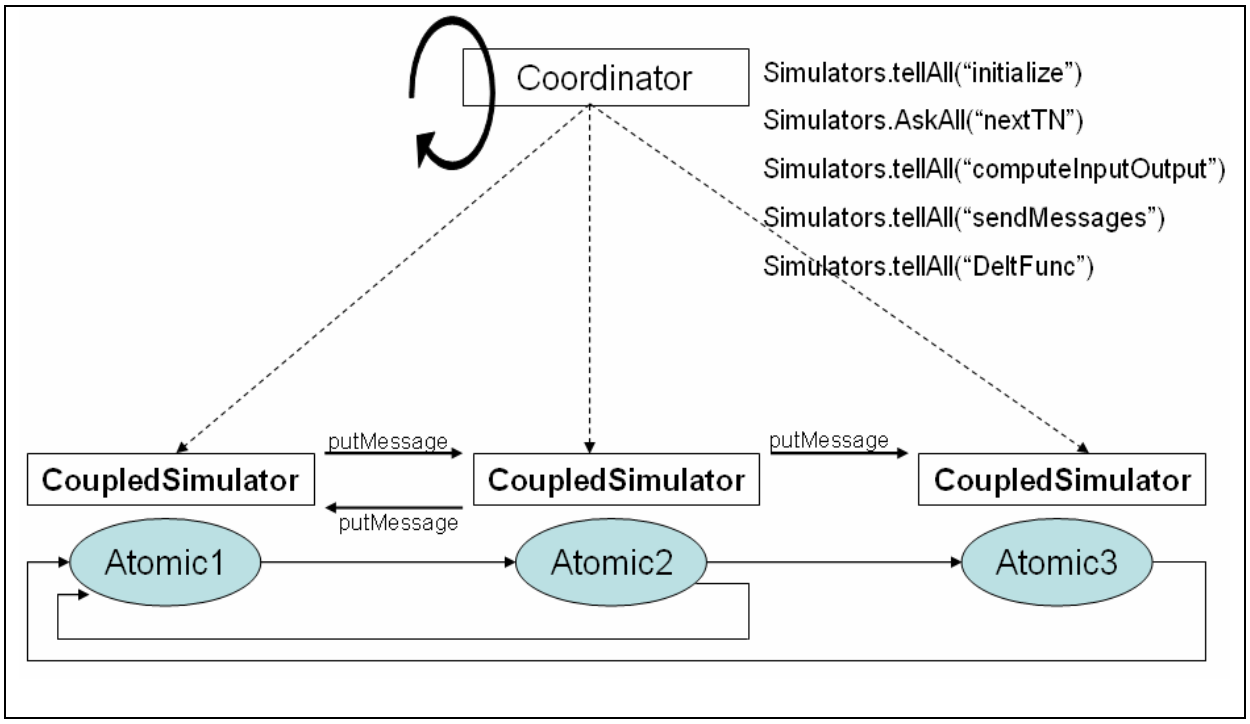
Existe, porém, uma construção simplificada do modelo, na qual todos os componentes são atômicos. Como não existe nenhum modelo *coupled* definido, o modelo é chamado de modelo *coupled* "*one-leve*l" (de um nível só).

```
Devs-coordinator
  Variables:
    DEVN = (X, Y, D, {M_d}, {I_d}, {Z_i,d}, Select)
      // the associated network
    Parent          // parent coordinator
    tl              // time of last event
    tn              // time of next event
    event-list
      // listo f elements (d, tn_d) sorted by tn_d and Select
    D*              // selected imminent child
When receive i-message (i, t) at time t
  for-each d in D do
    send i-message (i, t) to child d
  sort event-list according to tn[d] and Select
  tl = max { tl_d | d ∈ D }
  tn = min { tn_d | d ∈ D }
When receive *-message (*, t) at time t
  if t != tn then
```

```
    error: bad synchronization
  d* = first (event-list)
  send *-message (*, t) to d*
  sort event-list according to tn_d and Selec
  tl = t
  tn = min{ tn_d | d ∈ D }
when receive x-message (x,t) at time t with external input x
  if not ( tl ≤ t ≤ tn)
    then error: bad synchronization
  // consult external input coupling to get children influenced by
  // the input
  receivers = {r | r ∈ D, N ∈ I_r, Z_N,r (X) ≠ Φ}
  for-each r in receivers
   send x-messages (X_r, t) with input value X_r = Z_N,r (X) to r
  sort event-list according to tn_d and Select
  tl = t
  tn = min { tn_d | d ∈ D }
when receive y-message (y_d*,t) with output y_d* form d*
  // check external coupling to see if ther is an external out-
  // put event
  if d* ∈ I_N & Z_d*,N (Y_d*) <> Φ then
    send y-message (Y_N, t) with value Y_N = Z_d*,N(Y_d*) to parent
  // check internal coupling to get children influenced by output Y_d* of d*
  receivers = {r| r ∈ D, d* E I_r, Z_d*,r (Y_d*) ≠ Φ}
  for-each r in receivers
   send x-messages (X_r,t) with input value X_r = Z_d*,r(Y_d*) to r
end Devs-coordinator
```

Quadro 2 – Pseudo-código especificando o comportamento do coordenador.

Conforme Hu (2002, p. 2, tradução nossa), ”para simular esse tipo de modelo *coupled*, é necessário um Coordinator para controlar o ciclo completo de simulação. Para cada modelo atômico existe um CoupledSimulator para tratá-lo”. A Figura 9 mostra a relação existente entre o coodenador CoupledSimulator, o modelo *coupled* e seus componentes atômicos.



Fonte: Hu (2006, p.6)

Figura 9 - Ciclo de simulação de modelo do tipo *"one-level"*.

Como demonstra a Figura 9, existe um modelo CoupledSimulator para cada modelo atômico. O coordenador tem a tarefa de mandar todos os CoupledSimulator inicializarem, enviar seus tempos de próximo evento (nextTN), executar computeInputOutput(), sendMessages() e, ao final do ciclo, executar o método DeltFunc para a transição do modelo.

Para cada modelo de simulação há um componente denominado root-coordinator o qual é responsável por controlar o sistema como um todo. O Quadro 3 apresenta o pseudo-código que descreve o comportamento de um componente root-coordinator.

```
Devs-root-coordinator
  Variables:
    t       // current simulation time
    child   // direct subordinate devs-simulator or –coordinator
  t = t0
  send initialization message (i, t) to child
  t = tn of its child
  loop
       send (*, t) message to child
       t = tn of its child
  until end of simulation
end devs-root-coordinator
```

Quadro 3 – Pseudo-código especificando o comportamento de um root-coordinator.

A próxima seção apresenta resumidamente alguns conceitos associados ao contexto de multiprogramação visto ser este o contexto do problema a ser simulado.

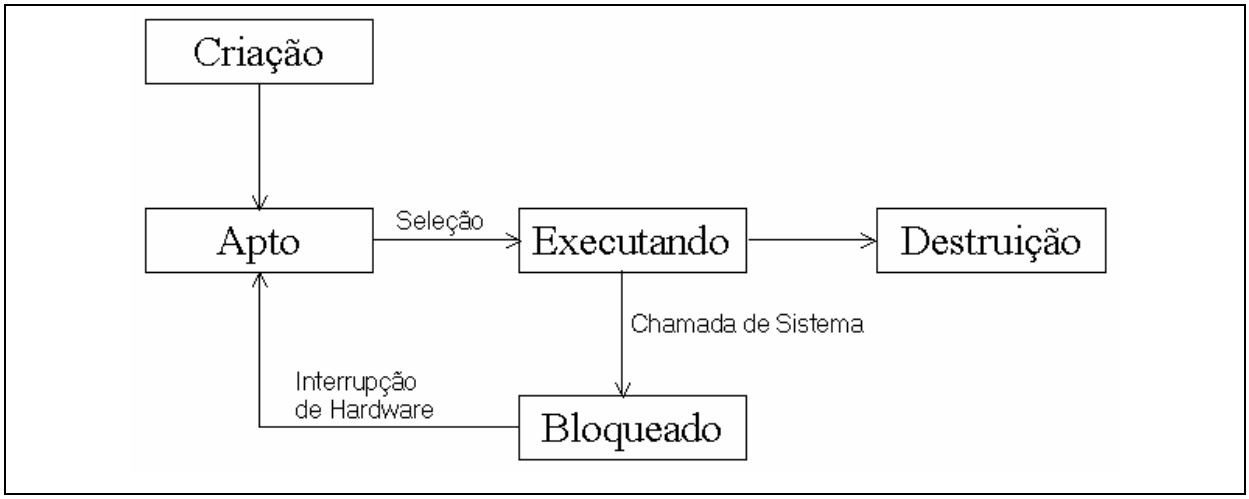
2.4 MULTIPROGRAMAÇÃO

Segundo Oliveira, Carissimi e Toscani (2000), mesmo em sistemas com um único processador, existem razões para o uso da programação concorrente em determinados tipos de aplicações. Os recursos do sistema operacional são maximizados com o uso da multiprogramação. Conforme Oliveira, Carissimi e Toscani (2000), na multiprogramação "diversos programas são mantidos na memória ao mesmo tempo". Nesse contexto, um programa é conceituado como uma seqüência de instruções. Um processo por sua vez, representa um programa em execução. Um programa pode ser executado por diversos processos ao mesmo tempo.

Durante sua execução, um programa não altera seu estado com o passar do tempo. Por outro lado, quando um processo entra num ciclo de processador, ou seja, no momento em que ele deseja ocupar o processador, tem seu estado alterado durante sua execução.

Segundo Oliveira, Carissimi e Toscani (2000, p. 17), "a mudança de estado de qualquer processo é iniciada por um evento. Esse evento aciona o sistema operacional, que então altera o estado de um ou mais processos". O sistema operacional controla a transição entre os estados do processo através de filas, as quais identificam os processos e seus estados.

A Figura 10 mostra os estados possíveis de um processo.



Fonte: Oliveira, Carissimi e Toscani (2000, p. 17)

Figura 10- Estados de um processo

Conforme Tanenbaum (2003, p. 142), “o uso da multiprogramação pode melhorar a utilização da unidade central de processamento (CPU)”. A medição dessa utilização resulta no grau da multiprogramação, estabelecido em função do tempo de utilização da CPU por um processo.

Na próxima seção serão apresentados alguns trabalhos correlatos à presente proposta.

2.5 TRABALHOS CORRELATOS

Durante o processo de levantamento bibliográfico para a fundamentação da presente proposta, foram encontrados diversos textos com citações, explanações e aplicações práticas do formalismo DEVS. A partir desses textos foi possível identificar as informações mais relevantes para a construção e aplicação de um simulador utilizando o formalismo proposto.

Foram identificados alguns trabalhos desenvolvidos utilizando-se DEVS como *framework* formal de modelagem, listados a seguir:

a) DEVS-JAVA: provê uma interface gráfica para a descrição dos objetos pertinentes à simulação de eventos discretos. Esta ferramenta foi desenvolvida com o intuito de aplicar a metodologia proposta por seu autor, demonstrando graficamente sua modelagem (ZEIGLER E SARJOUGHIAN, 2002);

b) CELL-DEVS: o paradigma Cell-DEVS permite a descrição de modelos celulares através de sua definição como modelos atômicos DEVS com vários tipos de atrasos. Cada célula será definida como um modelo atômico e poderão utilizar-se de diversas classes de atrasos para seu comportamento (WAINER, 2000);

c) POWERDEVS: esta ferramenta permite a criação de modelos DEVS básicos. Caracterizando-se por sua interface gráfica para edição do modelo simulado (PAGLIERO; LAPADULA E KOFMAN, 2003);

d) JDEVS: é uma ferramenta para modelagem e simulação de modelos de sistemas naturais. Estes são modelos cujo objetivo é simular o comportamento de um ecossistema para estudar novas interações. Esta ferramenta traduz a aplicação específica em que o formalismo DEVS pode ser focado (FILLIPI; CHIARI E BISGAMBIGLIA, 2002);

e) ambiente de animação de processos: através deste trabalho, é construído um ambiente para a animação de processos concorrentes, dessa forma contribuindo

> para a compreensão do funcionamento dos mecanismos dos sistemas operacionais (SANTOS, 2005).

Por ser um assunto pouco usado em entidades de ensino nacionais, a documentação referente ao formalismo DEVS utilizada neste trabalho é proveniente de sítios estrangeiros, *papers* e outros documentos disponibilizados por autores que utilizam-se do *framework* focado neste trabalho para o desenvolvimento de pesquisas no campo da modelagem e da simulação.

Os trabalhos citados são ditos correlatos a este trabalho pois implementam o *framework* DEVS para modelagem formal. O ambiente de simulação de processos concorrentes, desenvolvido por Santos (2005), tem correspondência com o estudo dos processos concorrentes proposto por este trabalho.

3 DESENVOLVIMENTO DO SISTEMA

Neste capitulo serão apresentados a estrutura do sistema e os componentes utilizados para a implementação da solução. Inicialmente são relacionados os componentes utilizados e posteriormente são apresentados os módulos professor e aluno.

3.1 REQUISITOS DO SISTEMA

A ferramenta para simulação deverá conter os seguintes Requisitos Funcionais (RF) e Não Funcionais (RNF), de acordo com os quadros apresentados abaixo.

O Quadro 4 apresenta os requisitos funcionais previstos para o sistema e sua rastreabilidade, ou seja, vinculação com os casos de uso associados.

Requisitos Funcionais	**Caso de Uso**
RF01: O sistema deverá permitir ao usuário informar os parâmetros do sistema para simulação	UC01
RF02: O sistema deverá permitir ao usuário executar o procedimento de simulação e obter a relação dos eventos ocorridos durante a simulação	UC02

Quadro 4- Requisitos funcionais

O Quadro 5 apresenta a lista dos requisitos não funcionais previstos para o sistema.

Requisitos Não Funcionais
RNF01: Deverá ser utilizada a ferramenta Delphi, versão 7.0, para a implementação das interfaces.
RNF02: O sistema deverá utilizar recursos de orientação a objetos.
RNF03: As informações de entrada e saída devem ser armazenadas em arquivo modo texto.
RNF04: O sistema deverá permitir que o modelo para simulação seja compatível com as especificações DEVS.

Quadro 5 - Requisitos não funcionais.

3.2 DIAGRAMA DE CASOS DE USO

Na Figura 11 é apresentado o diagrama de casos de uso e em seguida no Quadro 6 são apresentados os detalhamentos dos casos de uso.

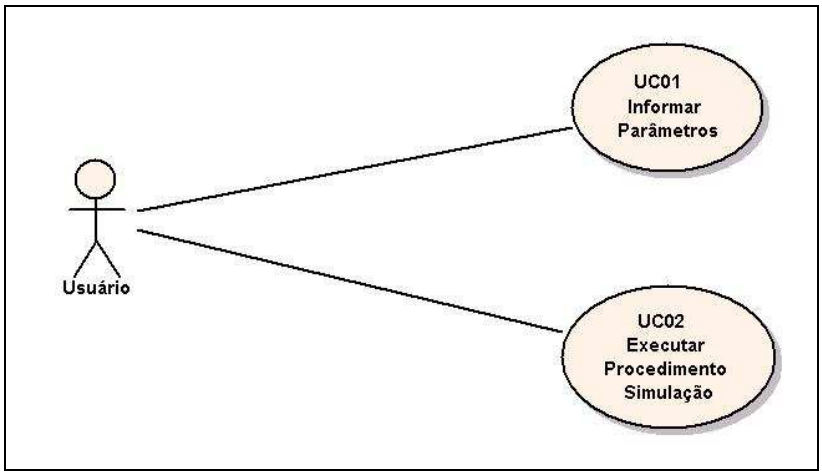


Figura 11 - Diagrama de casos de uso [UC01 e UC02]

O Quadro 6 detalha os casos de uso identificados.

UC01	Informar parâmetros
Objetivo	Permite ao usuário informar a parametrização de variáveis de controle vinculadas ao modelo atômico.
UC02	Executar Procedimento de Simulação
Objetivo	Permite ao usuário iniciar a simulação, dados os parâmetros de entrada e obtendo-se os parâmetros de saída. Após acessar a tela de simulação é feita a sequência dos eventos e a partir deste a *engine* DEVS produzirá o *log* de saída caracterizando aquele cenário de simulação.

Quadro 6 – Casos de Uso identificados

A próxima seção descreve a arquitetura do modelo construído.

3.3 A ARQUITETURA DO SISTEMA

A seguir são descritas as classes que implementam o framework de simulação de acordo com o modelo conceitual apresentado no capítulo anterior.

O diagrama de classes apresentado na Figura 12 demonstra uma visão global do sistema implementado. Um diagrama detalhado é apresentado na Figura 30 (pág. **Erro! Indicador não definido.**).

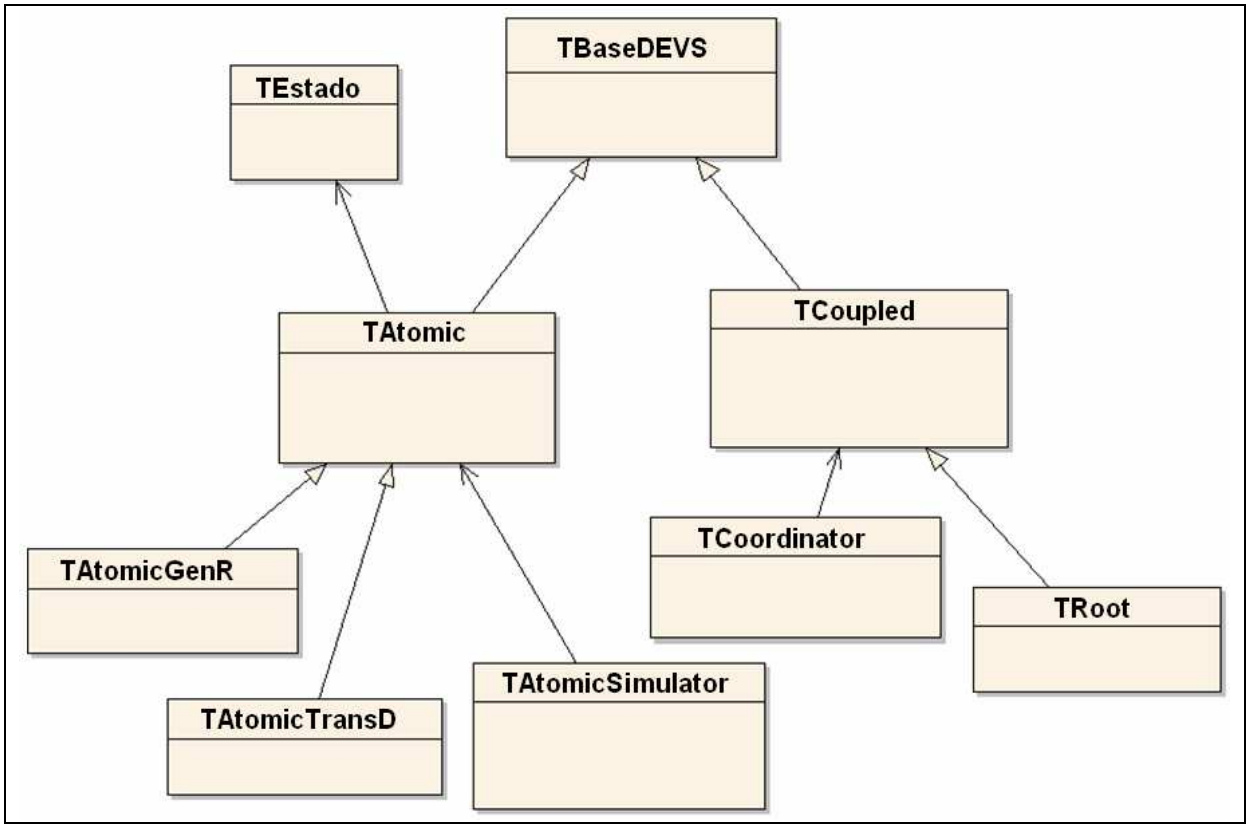


Figura 12 – Diagrama de classes do modelo implementado.

A classe TBaseDevs, descrita no Quadro 7, concentra os atributos que serão herdados pelos modelos atômico e coupled. Nesta classe estão descritas as listas de portas além dos métodos para criar, incluir, recuperar e manipular as portas de conexão dos modelos.

```
TBaseDEVS = class(TObject)  //Abstract base class for AtomicDEVS and CoupledDEVS classes.
Private
  prvID: integer;               // prvID é o identificador único do modelo Devs, consiste
                                // de um número sequencial, dado no momento da criação da
                                // TreeView
  prvModelName: String;         // denomina o modelo
  prvModelType: TTipoModelo;    // identifica o modelo conforme o tipo

  prvListaPortaIN: TList;       // são listas de referência para as portas de entrada e
  prvListaPortaOUT: TList;      // saída do modelo DEVS

  prvParent : TObject;          // Nesta implementação, o prvParent continua representando
```

```
                                  // o modelo ascendente ao atual
                                  // O "parent" de um Coupled é um Coordinator, de um Atômico
                                  // é um AtomicSimulator. O Root não tem modelo "parent"

    prvElapsedTime : Integer;     // tempo passado desde a última transição

    Function BuscaPorta(prmIDPorta: integer): Pointer;

  public

    Property FID: integer read prvID write prvID;
    Property FNomeDoModelo: string read prvModelName write prvModelName;
    Property FTipoDoModelo: TTipoModelo read prvModelType write prvModelType;
    Property FParent: TObject read prvParent write prvParent;
    Property FElapsedTime: integer read prvElapsedTime write prvElapsedTime;

    Function toString: String; overload;

    Procedure CriaListaPortaEntrada;
    Procedure CriaListaPortaSaida;

    Procedure AddPortaEntrada(prmPorta:TPorta);
    Procedure AddPortaSaida(prmPorta:TPorta);

    Function GetNroPortaEntrada: Integer;
    Function GetNroPortaSaida: Integer;

    Function GetListaPortaEntrada: TList;
    Function GetListaPortaSaida: TList;

    Function GetPortaSaida( prmNomePorta: String ): TPorta;
    Function GetPortaEntrada( prmNomePorta: String ): TPorta;

    Procedure AddPortaDestino(prmPortaOrigem: Integer; prmPortaDestino: Integer);
    Procedure CriaListaPortaDestino( prmPortaOrigem: integer );
    Function GetNroPortaDestino( prmPortaOrigem: integer ): integer;

    Function TempoToString(prmTempo: integer): string;

    Function GetElapsed: integer;
    Function IncrementaTempo( prmTempo: integer; prmTempoIncremento: integer ): integer;

    Procedure SetParent(prmMyParent:TObject);

  end;
```

Quadro 7 – Classe TBaseDevs

A classe TAtomic (Quadro 8) descreve a implementação de atributos e métodos do modelo básico atômico. Nesta classe estão definidos os métodos pertinentes ao modelo atômico DEVS: TimeAdvance, Saída, FuncaoInterna e FuncaoExterna.

Segundo Cantù (2003, p.51, grifo do autor), “as linguagens de POO permitem o uso de outra maneira de ligação, conhecida como *ligação dinâmica* ou *tardia* (*dynamic binding* ou *late binding*). Neste caso, o endereço real do método a ser chamado é determinado em tempo de execução com base no tipo da instância usada para fazer a chamada”. Ainda conforme Cantù (2003, p.53), “o polimorfismo significa que você pode chamar um método, aplicá-lo a uma variável, mas o método real que o Delphi chama depende do tipo de objeto ao qual a variável se relaciona”.

O recurso de polimorfismo de métodos é implementado utilizando-se as palavras-chave *virtual* e *override*.

```
Type
  TAtomic = class(TBaseDevs)
  private

    prvEstadoAtual: TEstado;     // ID do Estado atual do modelo

    prvListaEstado: TList;       // Lista de estados possiveis para o modelo

    prvSigma: integer;           //

  public

    Property FEstadoAtual: TEstado read prvEstadoAtual write prvEstadoAtual;

    Property FListaEstado: TList read prvListaEstado write prvListaEstado;


    constructor create;

    Procedure CriaListaEstados;
    Procedure AddEstado(prmEstado: TEstado);
    Function GetListaEstado: TList;
    Function GetNroEstados: Integer;

    Function CriaEstado(prmNomeEstado: string; prmTempo: integer): TEstado;
    Procedure SetEstadoAtual(prmEstado: TEstado);
    Function GetEstadoAtual: TEstado;
    Function getEstado(prmNomeEstado: string): TEstado;

    Function TimeAdvance: integer; virtual;
    function Saida: TMensagem; virtual;
    procedure FuncaoInterna; virtual;
    Function FuncaoExterna( prmElapsed: Integer; prmInputEvent: TMensagem): TEstado; virtual;

    Procedure HoldIn(prmPhase: TEstado; prmSigma: integer);
    Procedure SetSigma(prmSigma: integer);
    Function GetSigma: Integer;

    Procedure AtomicContinue( prmElapsed: Integer );
    Procedure Passivate;

    procedure Initialize; virtual;

  end;
```

Quadro 8 – Classe TAtomic

O Quadro 9 demonstra a implementação da classe TCoupled. Esta classe mantém uma lista de modelos associados à ela. O método AddSubModelo, o qual tem por objetivo adicionar modelos nessa lista, é implementado utilizando-se do recurso de *sobrecarga*. Nesse contexto, a classe pode conter métodos com o mesmo nome mas com listas de parâmetros diferentes e marcados com a palava-chave *overload* (Cantù, 2003, p38).

```
Type
  TCoupled = class(TBaseDevs)
  private

    prvListaSubModelos: TList;         // lista de modelos acoplados à este modelo

  public

    Procedure CriaListaSubModelos;
    Function GetNroSubModelos: integer;
    Procedure AddSubModelo(prmModelo:TAtomic); overload;
```

```
    Procedure AddSubModelo(prmModelo:TCoupled); overload;

    Procedure TrocaSubModelo(prmModeloOrigem: TObject; prmModeloDestino: TObject);

    Function GetListaSubModelos: TList;

  end;
```

Quadro 9 – Classe TCoupled

O Quadro 10 demonstra a implementação da classe TRoot.Responsável pela correta sincronização dos tempos de simulação, através do atributo CurrentTime.

```
Type
  TRoot = class(TCoupled)
  public
    CurrentTime: integer;      // Tempo atual da simulação

    Function Inicia(prmMensagem: TMensagem; prmRoot: TRoot; prmTempo: Integer): TStringList;
    Function Receive(prmPorta: Pointer; prmMensagem: TMensagem; prmFrom: TObject; prmTempo:
                     Integer): TStringList;

    Function SendAllMessages: TStringList;

  end;
```

Quadro 10 – Classe TRoot

O Quadro 11 demonstra a implementação da classe TCoordinator. Esta classe tem como objetivo tratar o envio e recebimento de mensagens durante o ciclo de execução da simulação. Nesta classe estão contidos o menor tempo para o próximo evento, implementado no atributo prvMenorTimeNext e, o maior tempo de execução do último evento, implementado no atributo prvMaiorTimeLast. A classe TCoordinator mantém uma lista de modelos iminentes, através do atributo prvListaIminente.

```
Type
  TCoordinator = class
  private
    prvCoupled: TCoupled;        // identifica o modelo coupled associado neste coordinator.
    prvListaIminente: TList;     // lista de componentes com seus tempos de próximo evento.

    prvMaiorTimeLast: integer;   // ultimo tempo de ocorrência de um modelo,
                                 // tl = max ( tl[d] | d E D )
    prvMenorTimeNext: integer;   // menor tempo entre os modelos iminentes,
                                 // tn = min ( tn[d] | d E D )

    prvTimeLast: integer;        // tempo em que foi executada a última transição
    prvTimeNext: integer;        // tempo da próxima execução
    prvElapsedTime: integer;     // tempo passado desde a última transição

    prvID: integer;               // Identificador Único do modelo
    prvModelName: String;         // denomina o modelo
    prvModelType: TTipoModelo;    // identifica o modelo conforme o tipo
    prvParent : TObject;          // Identifica o modelo ascendente (pai)

  public
```

```
    Property FCoupled: TCoupled read prvCoupled write prvCoupled;
    Property FListaIminente: TList read prvListaIminente write prvListaIminente;

    Property FID: integer read prvID write prvID;
    Property FNomeDoModelo: string read prvModelName write prvModelName;
    Property FTipoDoModelo: TTipoModelo read prvModelType write prvModelType;
    Property FParent: TObject read prvParent write prvParent;

    Property FTimeLast: integer read prvTimeLast write prvTimeLast;
    Property FTimeNext: integer read prvTimeNext write prvTimeNext;
    Property FElapsedTime: integer read prvElapsedTime write prvElapsedTime;

    constructor create;
    procedure Initialize;

    Property FMaiorTimeLast: integer read prvMaiorTimeLast write prvMaiorTimeLast;
    Property FMenorTimeNext: integer read prvMenorTimeNext write prvMenorTimeNext;

    Function Receive(prmMensagem: TMensagem;prmFrom: TObject; prmTempo: integer):TStringList;

    Procedure AddListaIminente(prmModelo: TBaseDevs);

    Procedure SendAllMessages;

    Procedure SendMessage(prmMensagem: TMensagem);

  end;
```

Quadro 11 – Classe TCoordinator

O Quadro 12 demonstra a implementação da classe TAtomicSimulator. Esta classe tem como objetivo receber e processar mensagens, através do método Receive, conforme o tipo de mensagem recebida, nas funções de transições do modelo. A associação entre a classe TAtomicSimulator e o seu modelo atômico é estabelecida através do atributo prvMyModel.

```
Type
  TAtomicSimulator =  class   //class( TObject)
   private

    prvMyModel: TAtomic;      // identifica o modelo atomico associado neste simulador

    prvID: integer;           // identificador único do modelo simulator
    prvTimeLast: integer;     // último tempo que o modelo atômico associado executou trans.
    prvTimeNext: integer;     // tempo que o modelo atômico associado irá executar transição
    prvElapsedTime: integer;  // tempo passado desde a última transição

    prvModelName: string;      // denomina o modelo simulator
    prvModelType: TTipoModelo; // tipo do modelo

    prvParent: TObject;

   public

     Property FID: integer read prvID write prvID;

     Property FMyModel: TAtomic read prvMyModel write prvMyModel;
     Property FTipoDoModelo: TTipoModelo read prvModelType write prvModelType;
     Property FParent: TObject read prvParent write prvParent;

     Property FTimeLast: integer read prvTimeLast write prvTimeLast;
     Property FTimeNext: integer read prvTimeNext write prvTimeNext;
     Property FElapsedTime: integer read prvElapsedTime write prvElapsedTime;
     Property FNomeDoSimulator: string read prvModelName write prvModelName;

     constructor create(prmMyModel: TAtomic);
     Procedure Initialize; overload; virtual;
     procedure Initialize(prmCurrentTime:integer); overload; virtual;
```

```
     function toString: String;

     function  Receive(prmMensagem:  TMensagem;  prmFrom:  TObject;  prmTempo:  integer  ):
TStringList;

     Function ComputeInputOutput(prmTempo: integer): TMensagem;

     Procedure SendMessage(prmMensagem: TMensagem );

   end;
```

Quadro 12 – Classe TAtomicSimulator

O Quadro 13 demonstra a implementação da classe TPorta. Uma porta define a associação entre os modelos ou *coupling*. Cada porta possui uma lista de portas de destino associada à ela. Através dessas portas destino serão enviadas as mensagens de saída para os modelos associados.

```
Type
  TPorta = class(TObject)
  private
    prvID: integer;              // Identificador da Porta
    prvOwner: TObject;           // Componente dono da porta - TObject para evitar referencia
                                 // circular com UBaseDevs
    prvTipoPorta: TTipoModelo;   // Tipo de porta

    prvNome: string;             // nome da porta
    prvTag: integer;             // campo auxiliar

    prvIDOwner: integer;         // identifica o Nodo ID ao qual a porta está diretamente
                                 // associada
    prvIDParent: integer;        // identifica o Nodo ID parent (limite de contexto)
    prvListaMensagem: TList;     // lista de mensagens na porta
  public

    FListaPortaDestino: TList;   // Associação para a porta destino (ver procedure
                                 // TfrmCoupling.btnAssociaPortaClick)
    Property FListaMensagem: TList read prvListaMensagem write prvListaMensagem;

    Property FID: integer read prvID write prvID;
    Property FOwner: TObject read prvOwner write prvOwner;
    Property FTipoPorta: TTipoModelo read prvTipoPorta write prvTipoPorta;

    Property FNomePorta: string read prvNome write prvNome;
    Property FTag: integer read prvTag write prvTag;
    Property FIDOwner : Integer read prvIDOwner write prvIDOwner;
    Property FIDParent : Integer read prvIDParent write prvIDParent;
    Function toString: String;

    Procedure putMessageOnPort( prmMensagem: TMensagem );
    Function getMessageOnPort( prmNomeMensagem: string ): TMensagem;
    Function temMensagemNaPorta: boolean;

    constructor create;

  end;
```

Quadro 13 – Classe TPorta

O Quadro 14 demonstra a implementação da classe TMensagem. A funcionalidade do mecanismo de simulação se dá através do envio e recebimento de mensagens entre os modelos. Esta classe contém os atributos utilizados para a comunicação entre os modelos.

```
Type
  TMensagem = class
  private
    prvValor: integer;            // Valor gerado pela função que enviou a mensagem
    prvTempo: Integer;            // Momento no Tempo em que foi gerada a mensagem
    prvNome: string;              // Nome do destino ou identificador da mensagem
  public
    FTipo : String[1];            // tipo mensagem i, *, X, Y
    Property FValor: integer read prvValor write prvValor;
    Property FTempo: integer read prvTempo write prvTempo;
    Property FNome: string read prvNome write prvNome;
  end;
```

Quadro 14 – Classe TMensagem

O Quadro 15 demonstra a implementação da classe TEstado. Um modelo atômico pode conter um ou vários estados. Esta classe define os atributos que serão utilizados na modelagem do estado no modelo atômico.

```
Type
  TEstado = class
  private
    prvID: integer;               // Identificador Único do modelo
    prvEstado: string;            // descrição do estado
    prvTempo: integer;            // tempo de vida do estado
  public
    Property FID: integer read prvID write prvID;
    Property FEstado: string read prvEstado write prvEstado;
    Property FTempo: integer read prvTempo write prvTempo;
  end;
```

Quadro 15 – Classe TEstado

O Quadro 16 demonstra a implementação da classe TAtomicoGenR, que é utilizada na construção do exemplo descrito no capítulo de estudo de caso.

```
Type
  TAtomicoGenr = class(TAtomic)
  private
    auxCount: integer;
    auxInt_arr_time: integer;

  public
    constructor create;

    Function TimeAdvance: integer; override;
    function Saida: TMensagem; override;
    procedure FuncaoInterna; override;
    Function  FuncaoExterna(  prmElapsed:  Integer;  prmInputEvent:  TMensagem):  TEstado;
override;

    Procedure Initialize; override;
  end;
```

Quadro 16 – Classe TAtomicoGenr

O Quadro 17 demonstra a implementação da classe TAtomicoTransD, a qual é utilizada na construção do exemplo descrito no capítulo de estudo de caso.

```
Type
  TAtomicoTransd = class(TAtomic)
  private
    auxClock: integer;
    auxArrived: integer;
    auxSolved: integer;

  public
    constructor create;

    Function TimeAdvance: integer; override;
    function Saida: TMensagem; override;
    procedure FuncaoInterna; override;
    Function  FuncaoExterna(  prmElapsed:  Integer;  prmInputEvent:  TMensagem):  TEstado;
override;

    Procedure Initialize; override;
  end;
```

Quadro 17 – Classe TAtomicoTransD

Ambas as classes citadas anteriormente têm como classe ascendente a TAtomic, da qual herdam todos os atributos e métodos. Dado a característica da implementação do formalismo DEVS, estas classes possuem atributos específicos ao problema modelado, bem como seus métodos principais TimeAdvance, Saída, FuncaoIntera e FuncaoExterna, sobrepõem os métodos da classe TAtomic, utilizando o recurso de *override.*

3.3.1 A LISTA DE IMINENTES

Um aspecto fundamental para a compreensão do funcionamento do formalismo DEVS está relacionado a correta implementação da chamada lista de iminentes. Esta lista contém elementos ordenados pelo menor tN e é utilizada pelo *coordinator* para ajustar o tN global do modelo de simulação.

A Figura 13 apresenta o modelo do *coordinator* destacando a lista de iminentes.

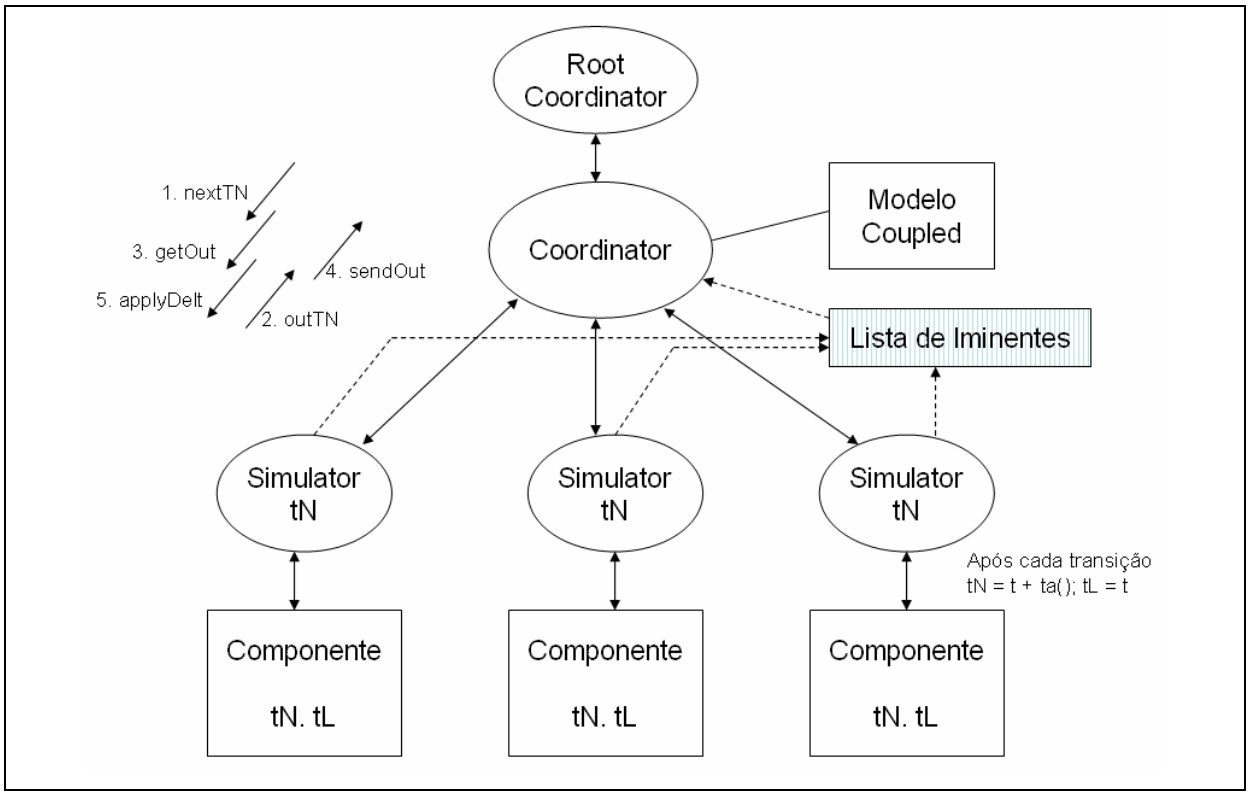


Figura 13 – Lista de iminentes no contexto de um coordinator

A próxima seção descreve as telas do sistema.

3.4 TELA PRINCIPAL

A Figura 14 apresenta a tela inicial do sistema onde é possível identificar os seguintes elementos:

1. Área de especificação do modelo a ser criado: um modelo de especificação consiste na identificação do número de elementos atômico e *coupled* conectados, essa facilidade identificada como item 1 da Figura 14 (atende ao caso de uso 01). Nesta área, o campo identificado como *coupled* permite informar o número de modelos *coupleds* que devem ser criados para o modelo a ser simulado enquanto o campo identificado como atômico informa o número de modelos atômicos que devem ser criados para o modelo. Da mesma forma, os campos identificados como Porta Entrada e Porta Saída, permite informar o número de portas associadas aos modelos.

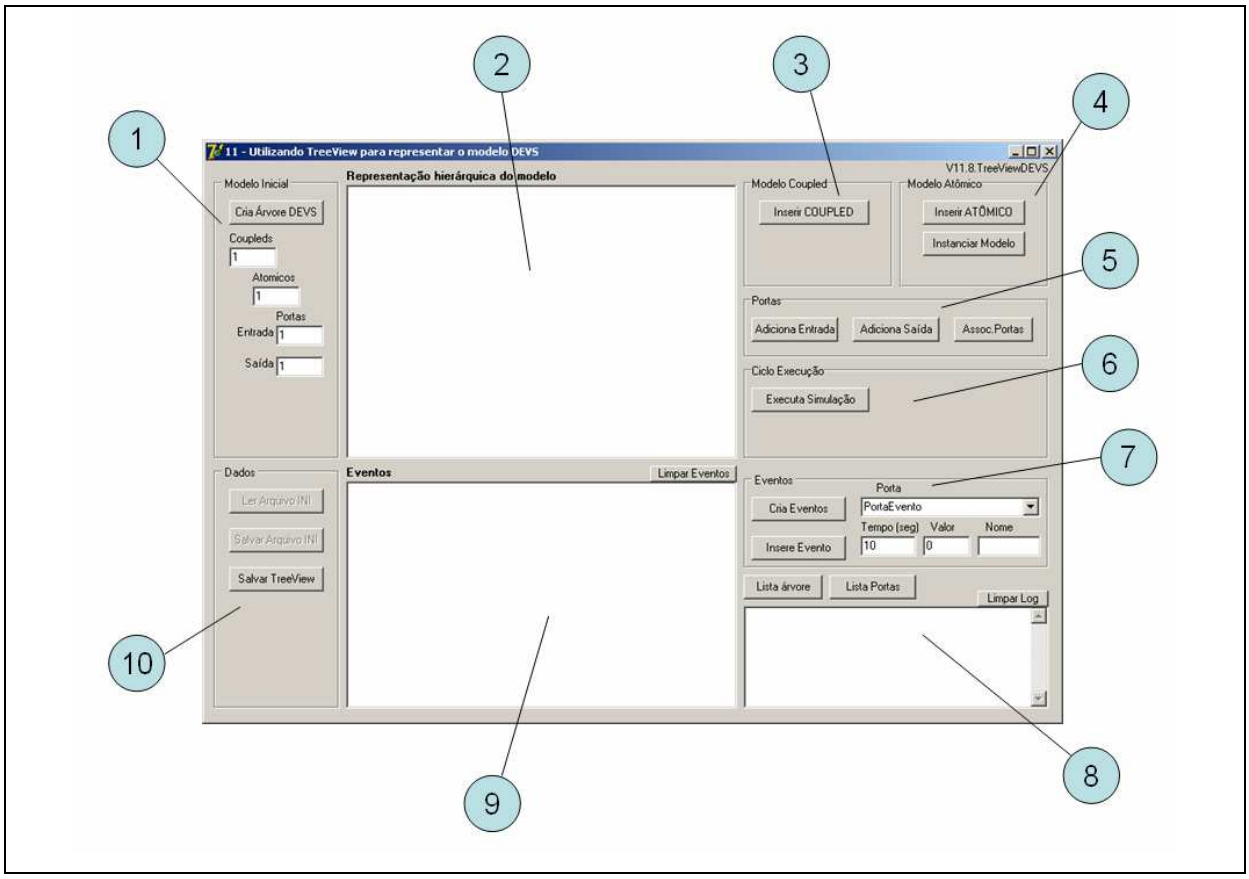


Figura 14 - Tela inicial do sistema

2. Área de representação da hierarquia: nesta área, a representação da árvore hierárquica DEVS é implementada utilizando o recurso de TreeView do Delphi (Figura 15). Com esse recurso, o sistema permite que sejam construídos modelos de forma que eles formem uma hierarquia e que, a partir dessa, seja feito o envio de mensagens entre os modelos.

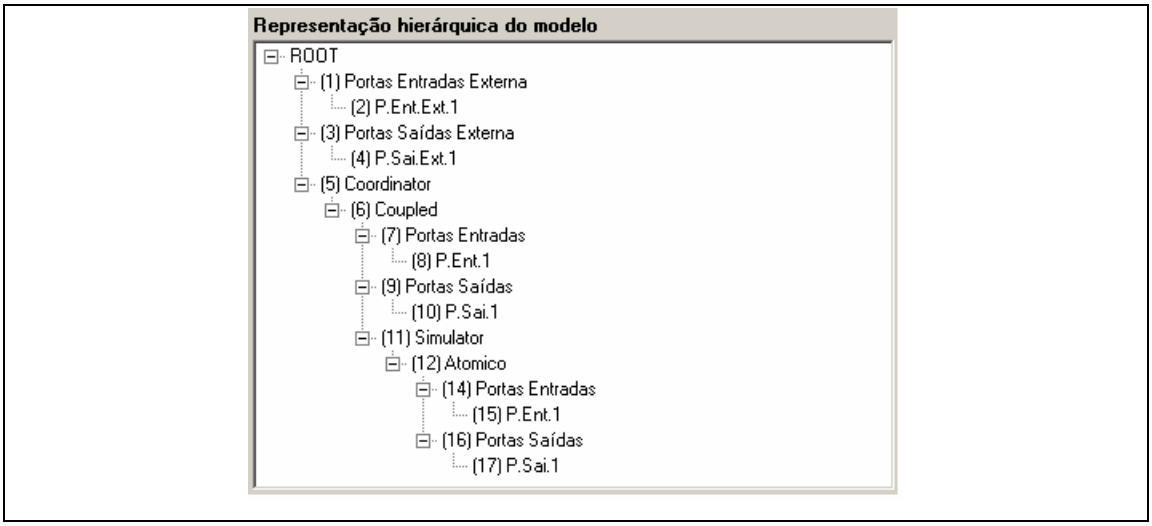


Figura 15 - Área de representação hierárquica do modelo.

3. Área de Modelo Coupled: o acionamento do botão “Inserir COUPLED” faz com que seja adicionado na estrutura um componente *coupled*. Essa facilidade está identificada como item 3 na Figura 14. Porém, como restrição estrutural definida pelo formalismo, um elemento coupled só pode ser descendente de um outro componente *coupled* ou do próprio componente *root*.
4. Área de Modelo Atômico: o acionamento do botão “Inserir ATÔMICO” faz com que seja adicionado na estrutura um componente atômico. Essa facilidade está identificada como item 4 na Figura 14. Como restrição estrutural definida pelo formalismo, um elemento atômico só pode ser descendente de um componente *coupled* ou do componente *root*. Para que seja possível instanciar modelos definidos pelo usuário, clica-se no botão “Instanciar modelo”.
5. Estabelecendo conexões entre as portas: a definição da estrutura hierárquica DEVS, construída a partir da árvore hierárquica permite que sejam adicionadas na estrutura dois tipos de porta: a de entrada, onde serão imputados os valores de entrada externos ou provenientes de transições internas, e a de saída, onde serão gerados os valores das transições internas. Adicionalmente nesta estrutura é permitido que as portas sejam ligadas entre si, fazendo com que os diversos componentes tenham conhecimento da rede estrutural para envio e recebimento de mensagens através dos seus componentes coordinators. Após efetuada a associação, a porta origem é demarcada com um sinal “=>” seguido de um número, o qual indica a quantidade de portas destinos existem associadas àquela porta. As facilidades de portas estão identificadas no item 5 da Figura 14. O acionamento do botão “Assoc.Portas” remete para a tela de associação entre as portas do modelo. A Figura 16 apresenta um *log* obtido a partir da especificação das associações entre as portas de um modelo de simulação.

```
-------Lista de portas:11-------
Origem:(2) P.Ent.Ext.1
Origem:(4) P.Sai.Ext.1
Origem:EF-START
 Destino=>GENR.START
Origem:EF-OUT
Origem:GENR.START
Origem:GENR.OUT
 Destino=>TRANSD.ARRIVED
Origem:TRANSD.ARRIVED
Origem:TRANSD.OUT
 Destino=>EF-RESULT
Origem:EF-IN
 Destino=>TRANSD.DONE
Origem:TRANSD.DONE
Origem:EF-RESULT
```

Figura 16 - Log de associação entre portas de entrada e saída dos modelos.

6. Iniciar simulação: permite que sejam enviadas mensagens diretamente ao modelo. Esta implementação identificada como item 6, da Figura 17 atende ao caso de uso 2. Cada botão definido gera uma seqüência de eventos e, ao final do ciclo, tem-se uma rodada completa de simulação.

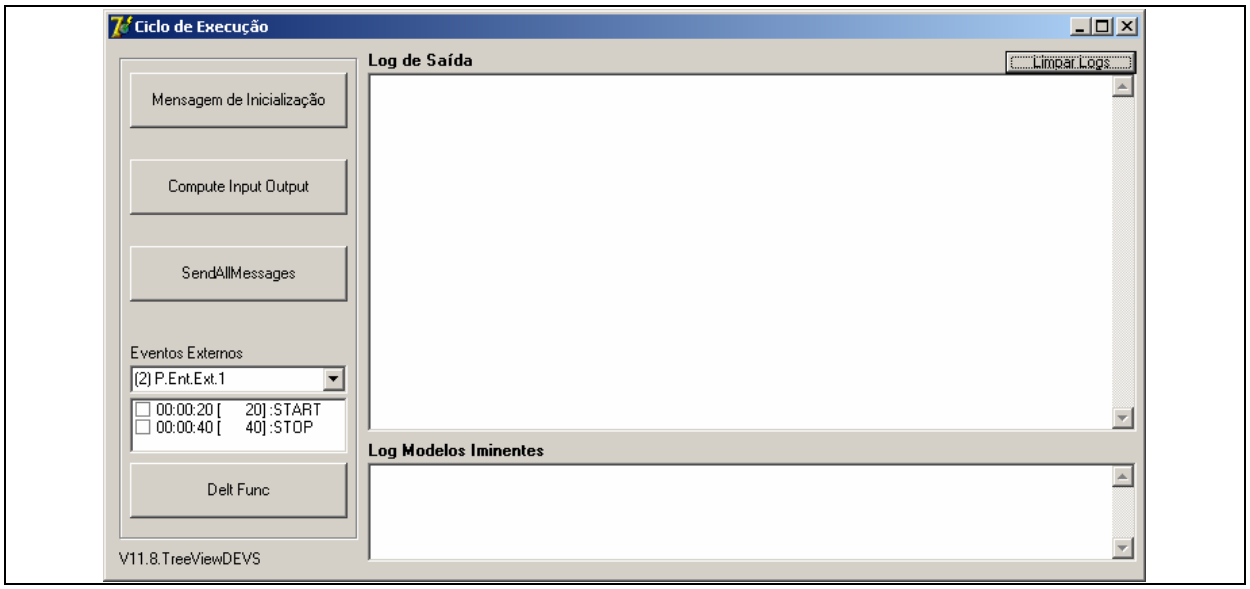


Figura 17 - Ciclo de Execução

7. Área de Geração de eventos: eventos são elementos externos ao sistema, aos quais são atribuídos tempo, valor e nome. O evento externo é enviado ao modelo conforme definido pelo campo tempo. Um evento está associado a uma porta de entrada externa, a identificação da porta é implementada por uma lista de portas onde é definida a porta externa a qual receberá os eventos externos. Cada evento tem um nome que o identifica unicamente dentro da função de transição externa, definida e implementada dentro do modelo atômico. Por esse motivo, o nome identificado no campo deve coincidir com o nome descrito internamente no código fonte da classe do modelo atômico. O botão “Insere evento”, adiciona o evento definido na lista de eventos. As facilidades que implementam a geração de eventos estão identificadas como item 7 da Figura 14.
8. Área de *log* de saída: este recurso é utilizado para listar e avaliar as conexões estabelecidas entre os modelos da árvore hierárquica DEVS. Dessa forma, é possível analisar, antes de submeter à simulação, a construção e ligação dos componentes. Esta facilidade está identificada como item 8 na Figura 14.

9. Área de seqüência de eventos: nesta área é implementada a lista de eventos. Antes de inserir evento, a lista de eventos deve ser criada. O botão “Criar Eventos” implementa essa funcionalidade. A lista de eventos pode ser apagada através do recurso “Limpar Eventos” disponibilizada nessa área.
10. Área de dados: implementa a funcionalidade de salvar a treeview em arquivo texto.

O próximo capítulo apresenta um estudo de caso do emprego do sistema na simulação de um modelo hierárquico.

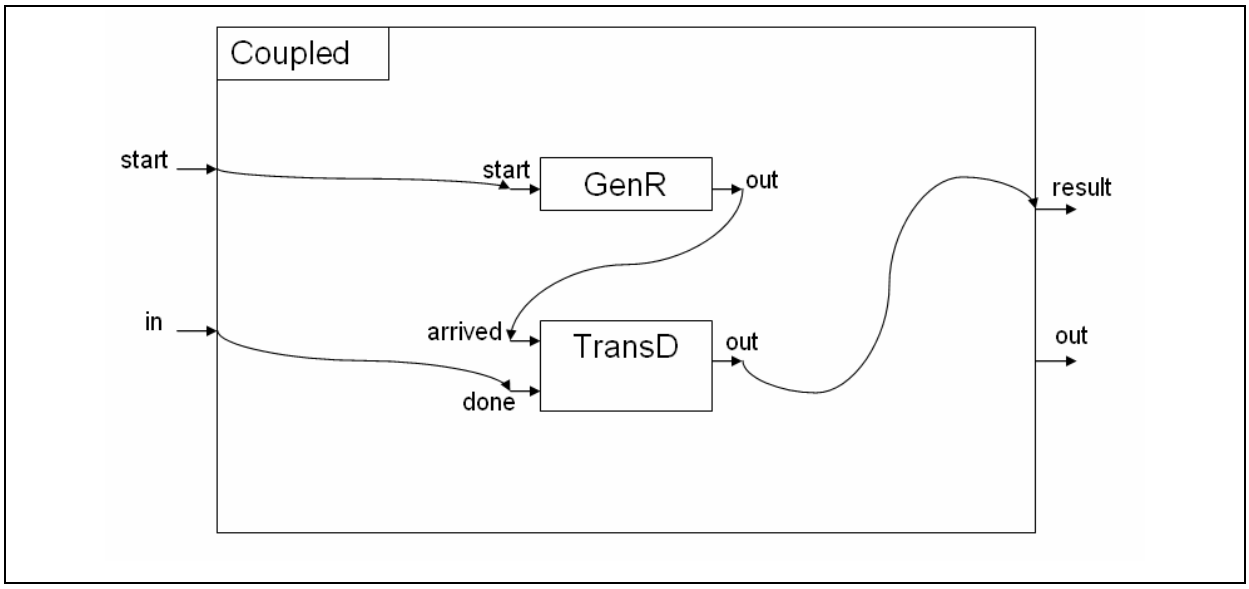
4 ESTUDO DE CASO

Este capítulo tem como objetivo demonstrar a operacionalidade e as funcionalidades do software desenvolvido. Para tanto foi criado um protótipo no qual é possível configurar os modelos e executar a simulação entre eles, de forma a obter um resultado dessa simulação, utilizando um exemplo apresentado em Ziegler (1990).

4.1 EXEMPLO DE MODELO DE SIMULAÇÃO

O modelo utilizado como exemplo possui dois componentes: um denominado *generator*, que tem por função produzir trabalhos, e outro denominado *transducer* que tem por função consumir os trabalhos produzidos pelo gerador. Estes trabalhos consistem em uma definição genérica de "job" a ser submetido a um processador. O funcionamento destes modelos é descrito a seguir.

A Figura 18 apresenta o problema a ser modelado.

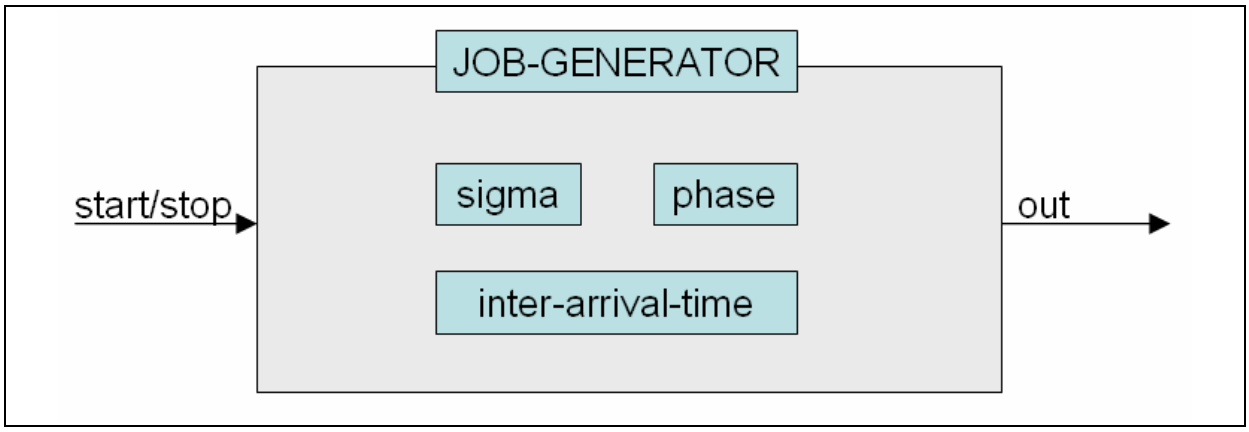


Fonte: Ziegler, Praehofer, Kim (2000, p.95)

Figura 18 – Modelo de simulação do estudo de caso

4.1.1 MODELO GENERATOR

O modelo *generator*, ou **GenR**, demonstrado na Figura 19, gera uma seqüência intermitente de tarefas (jobs). O mecanismo básico que produz este comportamento é a função de transição interna do modelo (Quadro 18).



Fonte: Adaptado de Ziegler(1990, pg.92)

Figura 19 – Modelo Generator (GenR)

Este procedimento faz com que o modelo retorne ao estado de "ACTIVE" após cada execução da transição interna e faz o agendamento para passar à próxima transição no tempo determinado por "*inter-arrival-time*" (intervalo entre eventos). Somente antes da transição interna acontecer, a saída de um identificador de "job" é produzida. Como o GenR é por si só um modelo DEVS, ele pode ser testado de forma isolada.

Em princípio, um *generator* é um modelo autônomo (seu comportamento é auto induzido através dos recorrentes eventos internos). Em função disto, ele não precisa de uma função de transição externa para definir sua resposta para eventos de entrada externos. Neste sentido, o modelo *generator* é imune a eventos externos.

O Quadro 18 apresenta o código-fonte que implementa o modelo *generator* caracterizando-o como um modelo descendente do modelo atômico conforme especificado no modelo geral DEVS.

```
unit UAtomicoGenr;

interface
uses
  Classes,
  UAtomic,
  UTipos;

Type
  TAtomicoGenr = class(TAtomic)      // Classe Generator
  private
    auxCount: integer;               // contador de jobs criados
    auxInt_arr_time: integer;        // Interval Arrive Time
  public
    constructor create;

    Function TimeAdvance: integer; override;
    function Saida: TMensagem; override;
    procedure FuncaoInterna; override;
    Function FuncaoExterna( prmElapsed: Integer; prmInputEvent: TMensagem): TEstado; override;

    Procedure Initialize; override;
  end;

implementation
```

```
uses
 UPorta;

{ TAtomicoGenr }

constructor TAtomicoGenr.create;
begin
  inherited create;

  FNomeDoModelo := 'AtomicoGenr';     // na criação da instância da classe serão...

  CriaEstado('ACTIVE',10);            //... setados os estados do modelo GenR
  CriaEstado('PASSIVE',INFINITY);
  CriaEstado('FINISHING',0);

end;

function TAtomicoGenr.FuncaoExterna(prmElapsed: Integer; prmInputEvent: TMensagem): TEstado;
var
  i: integer;
  auxMsg: TMensagem;
begin
  atomicContinue( prmElapsed );

  if FEstadoAtual.FEstado = 'PASSIVE' then     // ao receber uma mensagem e sendo o Estado PASSIVE,
     begin                                     // ...o modelo busca por...
       for i:=0 to self.GetNroPortaEntrada -1 do
         begin
           auxMsg :=TPorta(self.GetListaPortaEntrada.Items[i]).getMessageOnPort('START');
           if assigned( auxMsg ) then
              HoldIn(getEstado('ACTIVE'),auxInt_arr_time);   // ... uma mensagem de START para mudar o
         end;                                                 // ... estado para ATIVE e produzir jobs…
     end
  else
    if FEstadoAtual.FEstado = 'ACTIVE' then
       begin
         for i:=0 to self.GetNroPortaEntrada -1 do
           begin
             auxMsg :=TPorta(self.GetListaPortaEntrada.Items[i]).getMessageOnPort('STOP');
             if assigned( auxMsg ) then                    // ... ou uma mensagem de STOP para mudar o
                SetEstadoAtual(getEstado('FINISHING'));    // ... estado para FINISH e parar de produzir
           end;                                            // jobs.
       end;
  result := GetEstadoAtual;

end;

procedure TAtomicoGenr.FuncaoInterna;
begin
  if FEstadoAtual.FEstado = 'ACTIVE' then          // na função interna, o estado continua o mesmo.
     begin
       auxCount := auxCount + 1;
       HoldIn(getEstado('ACTIVE'), auxInt_arr_time);
     end
  else
     Passivate;
end;

procedure TAtomicoGenr.Initialize;                 // ao inicializar o modelo, o estado é setado para
begin                                              // ... ACTIVE

  inherited Initialize;

  HoldIn( getEstado('ACTIVE'), 10);
  auxInt_arr_time := 10;

end;

function TAtomicoGenr.Saida: TMensagem;           // a função de saída produz a mensagem ARRIVED, ou
var                                               // ... seja, o GenR produziu um job.
  auxMensagem: TMensagem;
begin
  auxMensagem := TMensagem.Create;
  auxMensagem.FValor := auxCount;
  auxMensagem.FNome  := 'ARRIVED';

  result := auxMensagem;
end;

function TAtomicoGenr.TimeAdvance: integer;
begin
  result :=  GetSigma;
end;

end.
```

Quadro 18 – Implementação do modelo GenR

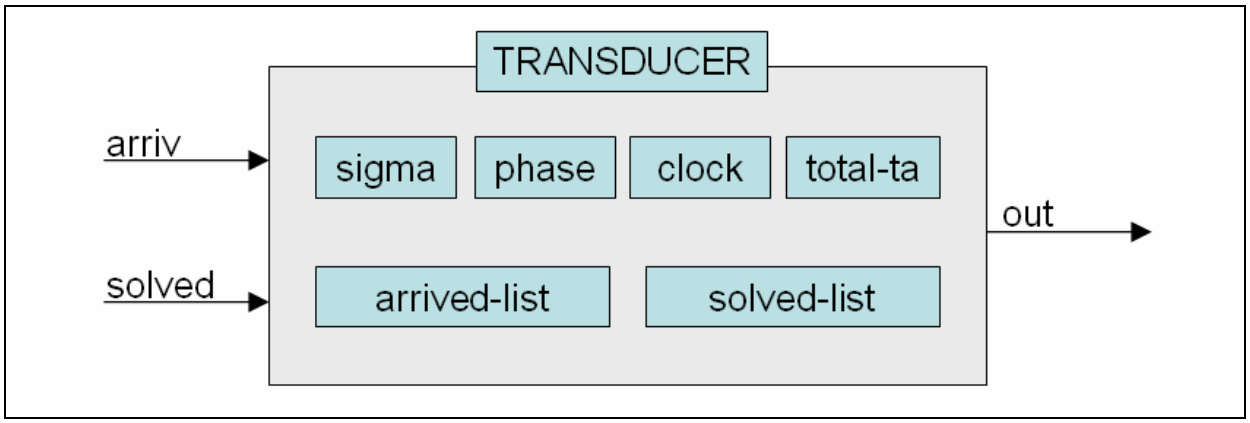
4.1.2 MODELO TRANSDUCER

O modelo *transducer*, ou **TransD**, demonstrado na Figura 20, tem como função medir dois índices de performance de interesse no contexto de simulação de processadores de computador: *throughput* (eficiência) e o tempo médio de *turnaroud* em uma rodada de simulação. Convém lembrar que o *throughput* refere-se a taxa média de *jobs* passados pela arquitetura, estimado pelo número de *jobs* processados durante o intervalo de simulação, divido pelo comprimento do intervalo. Um tempo de *turnaround* de um *job* é o comprimento de tempo entre sua chegada ao processador e sua passagem deste até ser completado (ou, o problema estar resolvido). Convém observar que para um processador simples P, o tempo de *turnaround* é o mesmo que o tempo de processamento. Contudo, para arquiteturas mais complexas este relacionamento não é necessariamente verdadeiro.

Para calcular as medidas de performance, o modelo TransD insere identificadores de jobs (Job id) que chegam na porta de entrada “ariv” (*arrival*) na sua lista de chegadas juntamente com seus tempos de chegada. Quando, e se, o job-id também aparece na porta de entrada dos resolvidos (“solved”), o modelo TransD coloca-os na lista de resolvidos e também calcula o seu tempo de *turnaround*.

O modelo TransD mantém o seu próprio relógio local para medir os tempos de chegada e *turnaround*. O formalismo DEVS não disponibiliza o tempo do relógio de simulação para modelos componentes. Dessa maneira os modelos têm que manter os seus próprios relógios de tempos se necessário. Segundo Ziegler (1990), isso pode ser implementado acumulando-se a informação de tempo decorrido presente no formalismo. Este tempo está disponível na forma das variáveis ***sigma*** e **e** (*elapsed time*).

Ao contrário do modelo GenR, um modelo TransD é essencialmente guiado pela sua função de transição externa. No modelo TransD uma transição interna é usada somente para causar uma saída ao final do intervalo de observação.



Fonte: Adaptado de Ziegler (1990, pg. 95)

Figura 20 – Modelo Transducer (TransD)

O Quadro 19 apresenta a implementação do modelo TransD conforme descrito acima, destacando-se que TransD também é descendente da classe TAtomic (Quadro 8) e, portanto, implementa o comportamento definido na arquitetura DEVS.

```
unit UAtomicoTransd;

interface
uses
  Classes,
  UAtomic,
  UTipos;

Type
  TAtomicoTransd = class(TAtomic)     // Classe Transducer
  private
    auxClock: integer;                // Relógio para controle de tempo dentro deste componente
    auxArrived: integer;              // Contador para a quantidade de jobs recebidos
    auxSolved: integer;               // Contador para a quantidade de jobs resolvidos
  public
    constructor create;
    Function TimeAdvance: integer; override;
    function Saida: TMensagem; override;
    procedure FuncaoInterna; override;
    Function FuncaoExterna( prmElapsed: Integer; prmInputEvent: TMensagem): TEstado; override;
    Procedure Initialize; override;
  end;

implementation
uses
 UPorta;

{ TAtomicoTransd }
constructor TAtomicoTransd.create;                //na criação da instância da classe serão...
begin                                             //...inicializadas as variáveis de controle e...
  inherited create;                               //...setados os estados do modelo TransD
  FNomeDoModelo := 'AtomicoTransd';
  FListaEstado := TList.Create;
  CriaEstado('ATIVE',50);
  CriaEstado('PASSIVE',INFINITY);
end;

function TAtomicoTransd.FuncaoExterna(prmElapsed: Integer; prmInputEvent: TMensagem): TEstado;
begin
  auxClock := auxClock + prmElapsed;
  AtomicContinue( prmElapsed );
  if prmInputEvent.FNome = 'ARRIVED' then        // para cada mensagem ARRIVED, é contado um job recebido
     auxArrived := auxArrived + 1
  else
    if prmInputEvent.FNome = 'SOLVED' then       // para cada mensagem SOLVED, é contado um job resolvido
       auxSolved := auxSolved + 1;
  result := getEstadoAtual;
end;

procedure TAtomicoTransd.FuncaoInterna;          // a função interna deixa o componente em estado PASSIVE
begin
  auxClock := auxClock + getSigma;
  Passivate;
```

```
end;

procedure TAtomicoTransd.Initialize;          // ao inicializar, o Estado inicial é setado para ATIVE
begin
  inherited Initialize;
  HoldIn( getEstado('ATIVE'), 500);
  auxClock  :=0;
  auxArrived:=0;
  auxSolved :=0;
end;

function TAtomicoTransd.Saida: TMensagem;   // a função de saída gera a mensagem contendo o resultado da
var                                         // ...divisão entre a quantidade de jobs recebidos pela
  auxMensagem: TMensagem;                   // ...quantidade de jobs solucionados, ou seja, a eficiência
begin                                       // ...produzida pelo modelo.
  auxMensagem := TMensagem.Create;
  if auxSolved > 0 then
    auxMensagem.FValor:= auxArrived div auxSolved
  else
    auxMensagem.FValor:= 0;
  auxMensagem.FNome := 'OUT';
  result := auxMensagem;
end;

function TAtomicoTransd.TimeAdvance: integer;   //função de avanço de tempo do componente TransD
begin
  result := getSigma;
end;
end.
```

Quadro 19 - Implementação do modelo Transd

4.2 MODELAGEM DO EXEMPLO

Nesta seção é apresentado como o modelo conceitual descrito na seção anterior foi modelado e simulado na ferramenta construída.

Para a implementação do estudo de caso, foram definidos dois modelos atômicos a saber:

- TransD: este modelo faz a medição de índices de performance dos jobs que estão chegando através das portas de entrada do modelo.
- GenR: gera como saída uma seqüência de jobs. Cada job é retransimitido para a classe TransD para que essa faça o seu processamento.

Além disso, existe o modelo *coupled* EF que define a associação entre aqueles modelos

A Figura 21 mostra a representação do modelo hierárquico. O primeiro passo na construção do modelo é a estruturação da árvore hierárquica DEVS, a qual é construída automaticamente com base nos parâmetros de entrada: número de componentes *coupled*, número de componentes atômicos, número de portas de entrada e de saída dos componentes.

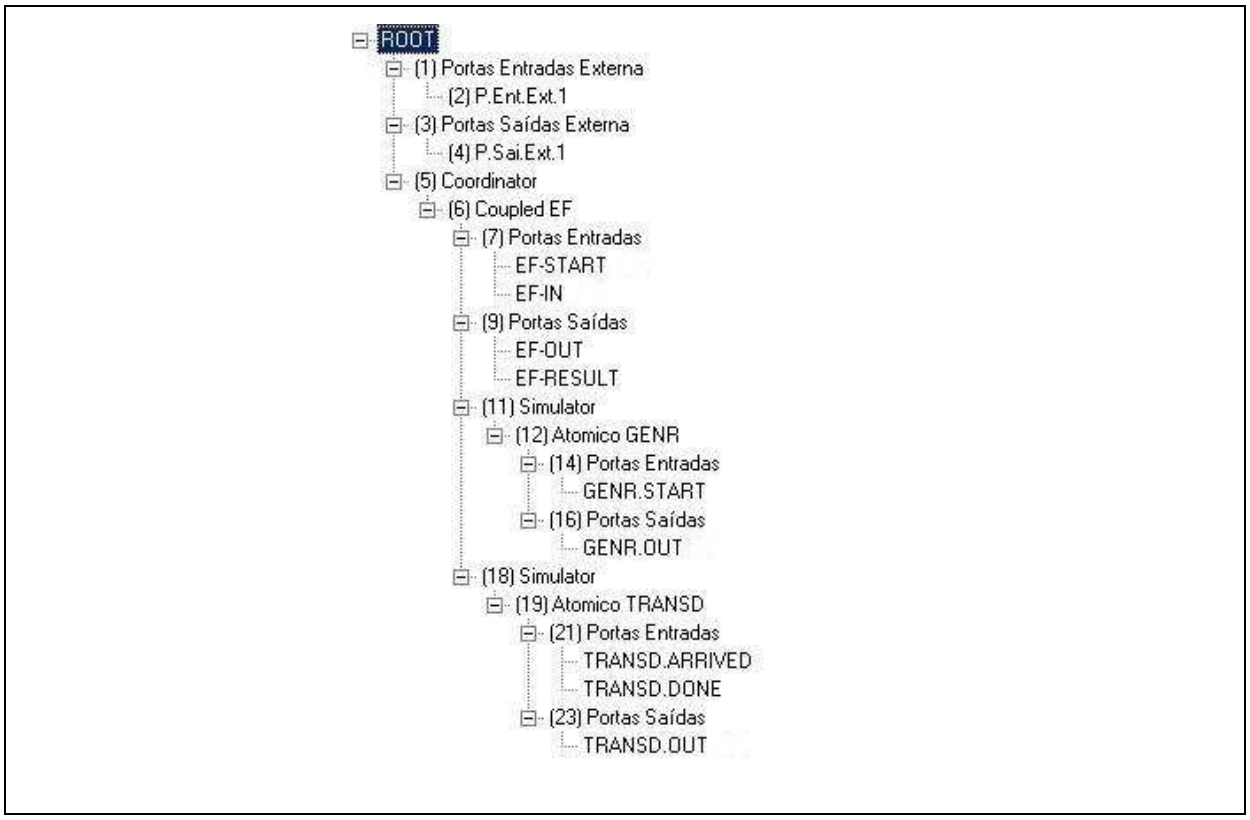


Figura 21 – Estrutura hierárquica do modelo simulado

4.2.1 ESPECIFICAÇÃO DO MODELO HIERÁRQUICO

A configuração do modelo hierárquico é implementada através da árvore de modelos. Com esse recurso, é possível estabelecer a condição de associação entre modelos (*coupling*) definida no formalismo DEVS. O modelo descreve a associação entre os modelos atômicos TransD e GenR com o modelo Coupled identificado na árvore hierárquica como “EF”.

4.2.2 ASSOCIAÇÃO DAS PORTAS

Uma vez criado o modelo, o próximo passo é realizar a associação entre as portas de entrada e saída dos modelos atômicos e *coupled* identificados.

Esta associação define o caminho válido entre as mensagens produzidas durante o processo de simulação.

A Figura 22 demonstra a associação de portas do modelo hierárquico, com a indicação de número de portas destino associadas.

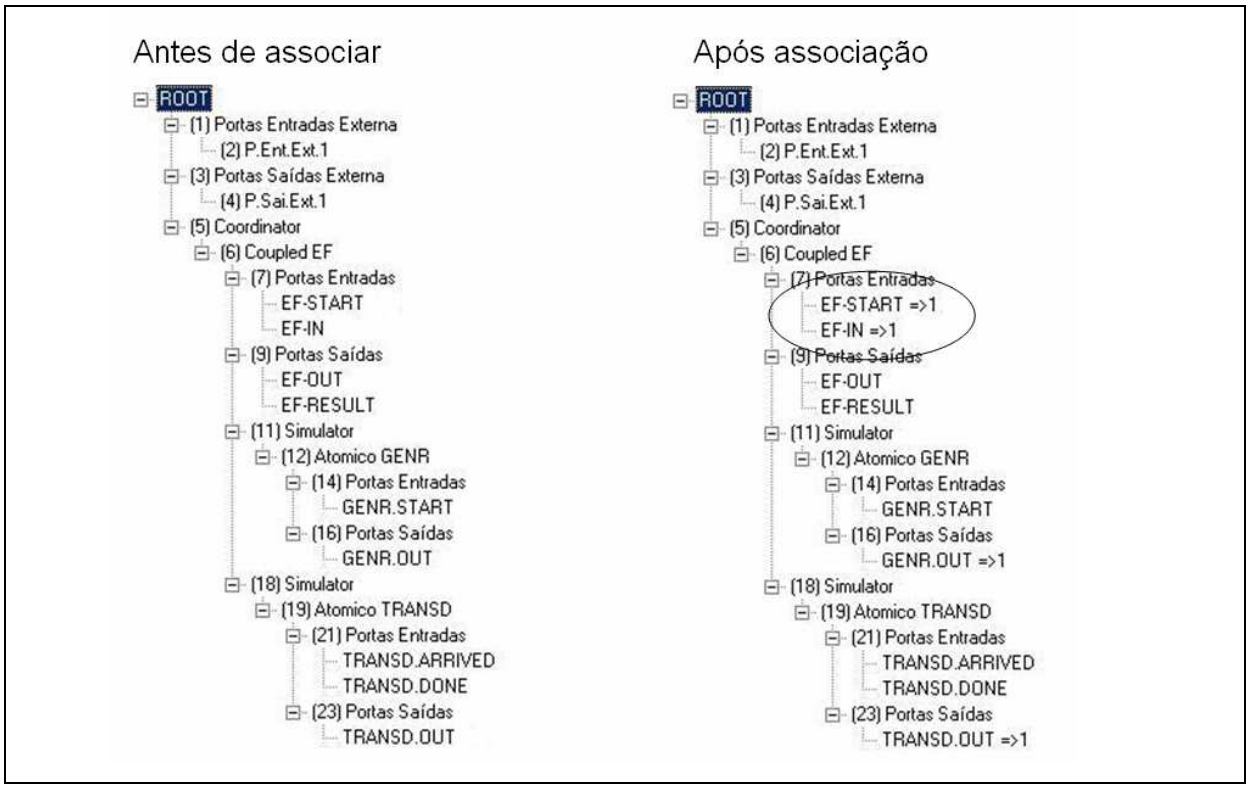


Figura 22 - Definição das portas de entrada e saída no modelo hierárquico.

A Figura 23 demonstra a associação entre a porta de origem “OUT” do modelo generator e a porta destino “ARRIVED” no modelo transducer.

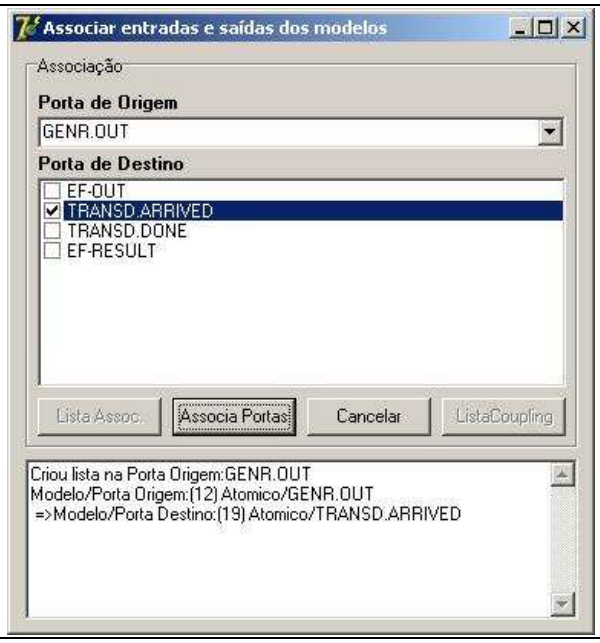


Figura 23 - Associação entre as portas de saída e entrada dos modelos GenR e TransD.

4.2.3 SUBSTITUIÇÃO DO MODELO GENÉRICO PELO MODELO ESPECÍFICO

A Figura 24 demonstra a associação da instância TAtomicoGenr ao modelo atômico da árvore hierárquica.

O *log* de saída demonstra as operações efetuadas pelo sistema durante a troca de contexto entre as instâncias. Sempre que o modelo é instanciado, sobrepondo o modelo anterior, essa substituição é refletida na árvore hierárquica.

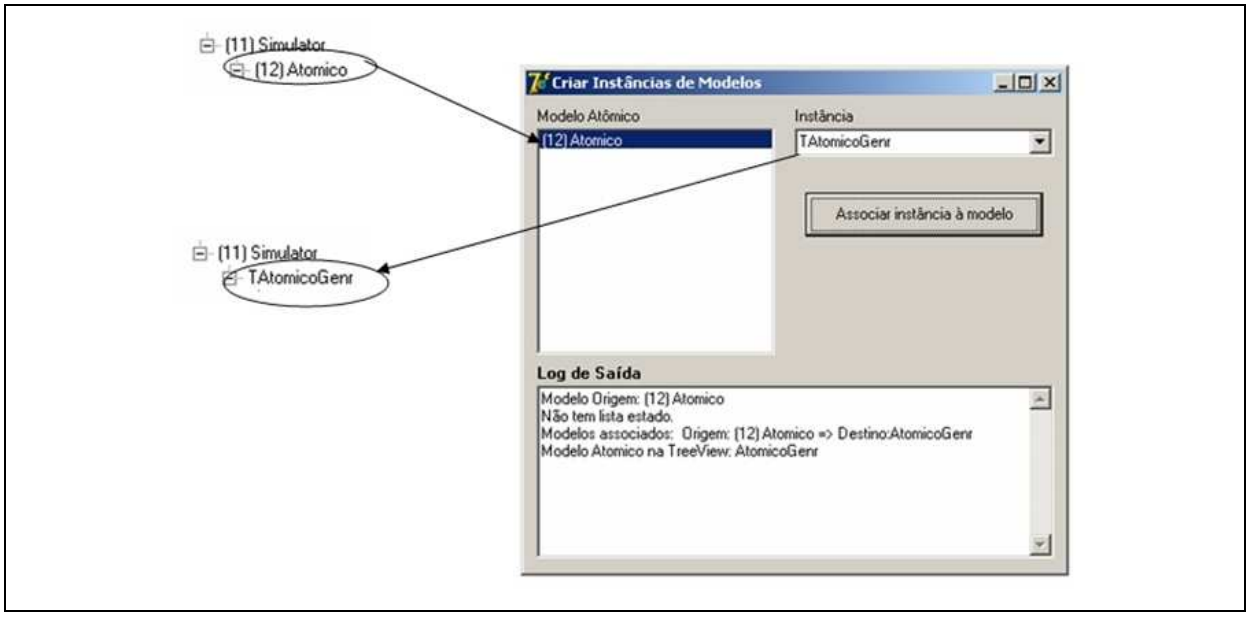


Figura 24 - Associar instância ao modelo.

4.2.4 CRIAÇÃO DOS EVENTOS

A Figura 25 demonstra a definição de uma seqüência de eventos externos para serem enviados ao modelo durante a sua simulação. Cada evento é identificado por um nome, “START” e “STOP” para o modelo GenR iniciar e parar a geração de jobs.

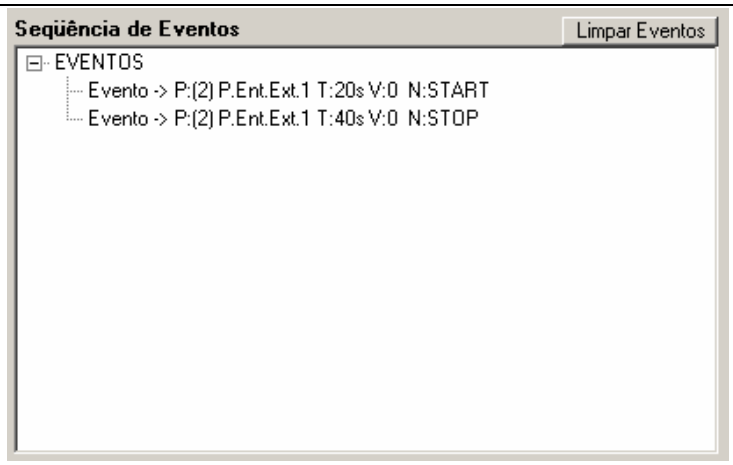


Figura 25– Definição da seqüência de eventos externos.

4.3 INICIO DA SIMULAÇÃO

Um ciclo de execução envolve inicializar os modelos, enviar uma mensagem para gerar as saídas (compute input/output), enviar as mensagens produzidas para os modelos conectados (sendAllMessages), inserir eventos externos e realizar a mudança de estados a partir das transições internas/externas (conforme estabelecido no formalismo).

4.3.1 INICIALIZAÇÃO DOS MODELOS

Ao iniciar, o modelo *root-coordinator* envia uma mensagem do tipo ‘I’ a todos os seus modelos subordinados para que estes iniciem seus tempos e indiquem qual o próximo tempo de evento interno. Nesta hierarquia de estudo de caso, o primeiro modelo a receber a mensagem é o *coordinator* do modelo *coupled*. Este vai repassar a mensagem de inicialização a cada modelo atômico associado a ele, através dos seus *simulators*. Ao receber a mensagem, o modelo atômico inicia seus tempos e devolve, em forma de mensagem, o tempo do seu próximo evento.

Ao receber os tempos de todos os modelos associados, o *coordinator* se encarrega de escolher o menor entre os tempos de próximo evento para definir qual o tempo do próximo evento do modelo C*oupled*. O *coordinator* repassa esta informação ao seu modelo *root-coordinator*, que detêm o sincronismo de tempos de eventos.

O modelo atômico que gerou a mensagem de tempo é inserido na lista de iminentes do coordinator, para indicar qual o modelo que executará no próximo tempo de evento.

Ao final dessa seqüência de inicialização, o modelo *root-coordinator* detêm a informação de qual o tempo de próximo evento (NextTN) da rodada de simulação seguinte.

A Figura 26 demonstra o envio da mensagem de inicialização a partir do *root-coordinator* a todos os seus modelos subordinados, retornando a informação de qual o menor tempo para o ciclo seguinte.

```
---- CICLO INICIALIZAÇÃO <i-message> ----
(R)  Mensagem:i/00:00:00 [       0] / ROOT  para (6) Coupled
(C)  Mensagem:i/00:00:00 [       0] / (6) Coupled para AtomicoGenr
(S)  Mensagem:D/00:00:10 [      10] / AtomicoGenr para (6) Coupled
(C)  Mensagem:i/00:00:10 [      10] / (6) Coupled para AtomicoTransd
(S)  Mensagem:D/00:08:20 [     500] / AtomicoTransd para (6) Coupled
(fC) Mensagem:D/00:00:10 [      10] / (6) Coupled para ROOT
=>NextTN: 00:00:10 [      10]
```

Figura 26 - Log de inicialização dos modelos.

4.3.2 COMPUTEINPUTOUTPUT

A função de saída é implementada dentro da função ComputeInputOutput na classe TAtomicSimulator, descrita na Figura 27.

A saída gerada pelo modelo atômico, através do método Saída, cria uma mensagem para o modelo *Simulator*, esta mensagem é do tipo 'Y' o que indica mensagem externa. Esta mensagem é depositada na porta de saída e ficará aguardando o próximo envio de mensagens entre os modelos para ser direcionada para o modelo destino.

```
Function TAtomicSimulator.ComputeInputOutput(prmTempo: integer): TMensagem;
var
  auxMensagem: TMensagem;
begin
  auxMensagem := TMensagem.Create;
  if prmTempo = FTimeNext then
    begin
      auxMensagem := FMyModel.Saida;
    end;
  auxMensagem.FTipo := 'Y';
  result := auxMensagem;
end;
```

Figura 27 - Implementação da função ComputeInputOutput.

4.3.3 SENDALLMESSAGES

Para que exista um sincronismo de mensagens de entrada e saída, entre os componentes do modelo de simulação, é necessário que um mecanismo faça o envio e recebimento das mensagens geradas durante o ciclo de simulação. Uma mensagem é gerada através de um método Saída, o qual produz uma mensagem na porta de saída do componente, definido na modelagem do modelo atômico. Esta mensagem poderá influenciar outro(s) componente dentro do modelo de simulação. A correta transmissão de mensagens entre as conexões de portas de entrada e saída, faz o modelo receber e tratar a mensagem conforme definido pela sua modelagem.

A Figura 28 demonstra a implementação do método SendAllMessagens pelo modelo coordinator.

```
Procedure SendAllMessages;
var
  auxListaSubModelos: TList;
  auxListaPortaDestino: TList;
  auxListaPortaSaida: TList;
  auxListaMensagem: TList;
  auxSubModelo: Pointer;

  i, m, d, e: integer;
begin

  { envia a mensagem para todos os modelos destino }
  auxListaSubModelos := TCoupled( FCoupled ).GetListaSubModelos;

  { para cada sub modelo }
  for i:=0 to auxListaSubModelos.Count-1 do
    begin
```

```
      auxSubModelo := auxListaSubModelos.Items[i];

      { para cada porta de saida do sub modelo }
      auxListaPortaSaida := TBaseDevs( auxSubModelo ).GetListaPortaSaida;

      for m:=0 to auxListaPortaSaida.Count-1 do
        begin

          { pega a lista de mensagens da porta atual...}
          auxListaMensagem := TPorta( auxListaPortaSaida.Items[m] ).FListaMensagem;

          { pega a lista de portas destino da porta atual...}
          auxListaPortaDestino:= TPorta( auxListaPortaSaida.Items[m] ).FListaPortaDestino;

          { distribui a mensagem para todas as portas destino}
          for e:=0 to auxListaMensagem.Count-1 do
            begin
              for d:=0 to auxListaPortaDestino.Count-1 do
                begin
                  TPorta(    auxListaPortaDestino.Items[d]    ).putMessageOnPort(    TMensagem(
auxListaMensagem.Items[e]) );
                end;

              { Remove a Mensagem da porta }
              auxListaMensagem.Delete( e );
            end
        end;
    end;
end;
```

Figura 28 - Implementação do procedimento SendAllMessages.

4.3.4 DELTFUNC

A função de transição externa injeta um evento externo no tempo de simulação t.

Ao receber o evento externo o *coordinator* envia uma mensagem do tipo “X” para cada modelo *simulator* associado para que ele faça o repasse da mensagem ao seu modelo atômico e este, por sua vez, faça o tratamento da mensagem conforme a definição de modelagem da função de transição externa. O Quadro 20 demonstra o *log* de execução da simulação após o modelo ter recebido um evento externo na fatia de tempo de 10 unidades de tempo.

```
      ---- CICLO EXECUÇÃO  <X-message>  ----

      (E)  00:00:10 [      10] porta:(2) P.Ent.Ext.1 valor:0 nome:STAT

      (R)  Mensagem:X/00:00:10 [      10] / ROOT  para (6) Coupled
      (C)  Mensagem:X/00:00:10 [      10] / (6) Coupled para AtomicoGenr
      (S)   Evento Externo: T=00:00:10 [      10] /  V=0 N=STAT
      (S)   FuncInt S->S: ACTIVE->ACTIVE
      (S)  Mensagem:D/00:00:10 [      10] / AtomicoGenr para (6) Coupled
      (fC) Mensagem:D/00:00:10 [      10] / (6) Coupled para ROOT
      =>NextTN: 00:00:10 [      10]
```

Quadro 20 – Log de execução de um evento externo.

4.4 CONSIDERAÇÕES FINAIS

Como pode ser observado o simulador construído permitiu a construção do modelo e sua simulação. Tendo em vista que o tempo necessário para o completo entendimento do modelo conceitual estrapolou o tempo previsto, não houve tempo hábil para a construção do modelo de simulação de processos concorrentes conforme apresentado na proposta do TCC.

Contudo, os modelos GenR e TransD apresentados constituem os elementos básicos para a construção do modelo proposto na medida em que o modelo GenR poderia ter seu nome modificado para GenClock (gerador de clock) representando as interrupções do relógio, que determinam a fatia de tempo a que um processo tem direito de executar e, o modelo TransD adaptado para mudar as funções de um escalonador de processos.

Dessa forma, o modelo conceitual do simulador de processos concorrentes ficaria como apresentado na Figura 29.

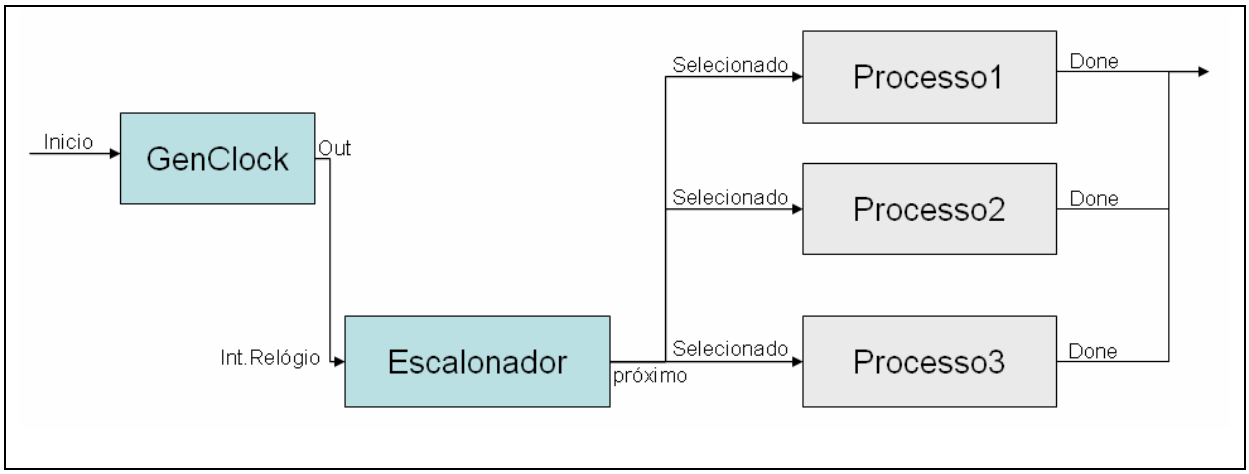


Figura 29 – Modelo de simulação de processos concorrentes utilizando DEVS.

5 CONCLUSÕES

O presente trabalho apresentou os fundamentos para a utilização de técnicas de desenvolvimento de um simulador de processos concorrentes utilizando uma metodologia forma de especificação de sistemas.

A simulação pode ser considerada uma maneira pela qual o comportamento de um sistema é observado, parametrizando-se os cenários estudados. A simulação a eventos discretos é modelada pelas definições das mudanças que ocorrem no tempo do evento.

O formalismo DEVS é um mecanismo de simulação a eventos discretos proposto por Zeigler e Sarjoughian (2002). Estabelece uma teoria de modelo de sistemas de tempo contínuo usando modelagem de eventos discretos. O formalismo define como gerar novos valores para as variáveis e os momentos que estes valores devem mudar.

Há que se destacar que, apesar de haverem várias implementações diferentes deste formalismo escritas em Java, C++ e Phyton, esta é a primeira implementação conhecida em Delphi, sendo que foi necessário construir toda uma arquitetura inexistente, sem a utilização de componentes previamente desenvolvidos.

O resultado obtido não superou as expectativas, todavia o modelo de *framework* viabilizado foi construído conforme especificação da literatura.

Todos os esforços foram concentrados no sentido de resolver um problema de sincronização de tempos de simulação. Suspeita-se que tal problema esteja na modelagem dos componentes GenR e TransD, pois o *framework* é genérico, conforme a especificação. Foram desenvolvidas mais de 20 versões do protótipo para corrigir a aquitetura do *framework*, sendo que as últimas versões foram implementadas no sentido de corrigir o avanço no relógio de controle de simulação.

Apesar de não atingir totalmente os objetivos propostos, o presente trabalho é significativo em relação ao formalismo DEVS, o que permite expandir horizontes de pesquisa em simulação de eventos discretos.

O alto nível de abstração da *engine* tornou este trabalho compatível com projetos de pesquisa desenvolvidos no curso de ciências da computação, o que colabora na dificuldade do tema de um trabalho para o curso de sistemas de informação.

5.1 LIMITAÇÕES

Embora a proposta original pretendesse validar o *framework* construído através da simulação do funcionamento de um ambiente de processos concorrentes, não foi possível atingir a este objetivo tendo em vista que a completa compreensão do formalismo demandou um esforço adicional o que acabou comprometendo o cronograma do projeto.

Ainda assim, foi descrito no texto uma proposta de modelo de ambiente de processos concorrentes utilizando os componentes construídos.

5.2 EXTENSÕES

Como extensões ao projeto sugerem-se:

a) automatização da seqüência de simulação sem a necessidade de execução passo-a-passo;
b) implementação de uma facilidade para armazenamento/recuperação de informações da árvore hierárquica no formato XML;
c) configuração dos modelos atômico e *coupled* em formato XML;
d) desenvolvimento de uma inteface que permita a especificação do modelo a ser simulado a partir de uma linguagem gráfica.

6 REFERÊNCIAS BIBLIOGRÁFICAS

CANTÙ, Marco. **Dominando o Delphi 7:** a bíblia. São Paulo : Pearson Education do Brasil, 2003.

FILIPPI, Jean-Baptiste; CHIARI, Frederic; BISGAMBIGLIA, Paul. Using JDEVS for the modeling and simulation of natural complex systems. In: AI, SIMULATION AND PLANNING IN HIGH AUTONOMY SYSTEMS, 12., 2002, Lisbon. **Proceedings**... Lisbon, 2002. Disponível em <http://spe.univ-corse.fr/filippiweb/publis/docs/ais.pdf>. Acesso em: 14 maio 2005.

FREITAS FILHO, P. J. **Slides de Aula – Introdução**. 2005. Disponível em <http://www.inf.ufsc.br/~freitas/cursos/simgrad/2005-2/Aulas/5101 A01/Slides Aula 1 - Introducao.pdf>. Acesso em: 07 set. 2005.

GIOZZA, William Ferreira. et al. **Redes locais de computadores**: protocolos de alto nível e avaliação de desempenho. São Paulo: McGraw-Hill, 1986.

HU, Xiaolin. **Simulation in DevsJava**. 2006. Disponível em <http://www.cs.gsu.edu/~cscxlh/CSC8910/DEVSJAVA_simulation.pdf>. Acesso em: 29.jun.2006.

MELLO, B. A. **Co-simulação distribuída de sistemas heterogêneos.** 2001. 99 f. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas operacionais**: série livros didáticos número 11. Porto Alegre: Ed. Sagra Luzzatto, 2000.

PAGLIERO, Estaban; LAPADULA, Marcelo; KOFMAN, Ernesto. **POWERDEVS.** Una herramienta integrada de simulación por eventos discretos. 2003. Disponível em <http://www.fceia.unr.edu.ar/~kofman/files/lsd0304.pdf>. Acesso em: 31 ago. 2005.

PRADO, Darci. **Teoria das filas e da simulação**. Belo Horizonte: Editora de desenvolvimento gerencial, 1999.

SANTOS, Marco Antonio Ruthes dos. **Animação do funcionamento de um núcleo de sistema operacional**. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2005.

SOARES, Luiz Fernando Gomes. **Modelagem e simulação discreta de sistemas**. Rio de Janeiro: Campus, 1992.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Tradução Ronaldo A. L. Gonçalves, Luis A. Consularo. 2. ed. São Paulo: Prentice Hall, 2003.

WAINER, Gabriel . **Cell Based Discrete-Event Simulation.** Information. 2000. Disponível em <http://www.sce.carleton.ca/faculty/wainer/celldevs/introduction.html>. Acesso em: 16 jan. 2005.

ZEIGLER, Bernard.P.; SARJOUGHIAN, Hessam S. DEVS Component-Based M&S Framework**:** An Introduction. In: AI, SIMULATION AND PLANNING IN HIGH AUTONOMY SYSTEMS, 12., 2002, Lisbon. **Proceedings**... Lisbon, 2002. Disponível em <http://www.acims.arizona.edu/EDUCATION/ACIMS DEVSTut AIS2002 Final.doc>. Acesso em: 15 jul. 2005.

ZEIGLER, Bernard.P.; SARJOUGHIAN, Hessam S. **Introduction to DEVS Modeling & Simulation with Java**: Developing Component-based Simulation Models. 2003. Disponível em <http://www.acims.arizona.edu/EDUCATION/ECE575Fall03/Manuscript_MSDJ_090903.pdf>. Acesso em: 15 jul. 2005.

ZEIGLER, Bernard. P. **Object-Oriented Simulation with Hierarchical, Modular Models**. Intelligent Agents and Endomorphic Systems. San Diego, CA: Academic Press, 1990.

ZEIGLER, Bernard. P; PRAEHOFER, Herbert; KIM, Tag Gon. **Theory of Modeling and Simulation**. Integratig Discrete Even and Continuous Complex Dynamic Systems. Second Edition. San Diego, CA: Academic Press, 2000.

7 APÊNDICE A – Diagrama de classes detalhado

A Figura 30 apresenta um diagrama de classes detalhado do *framework* implementado.

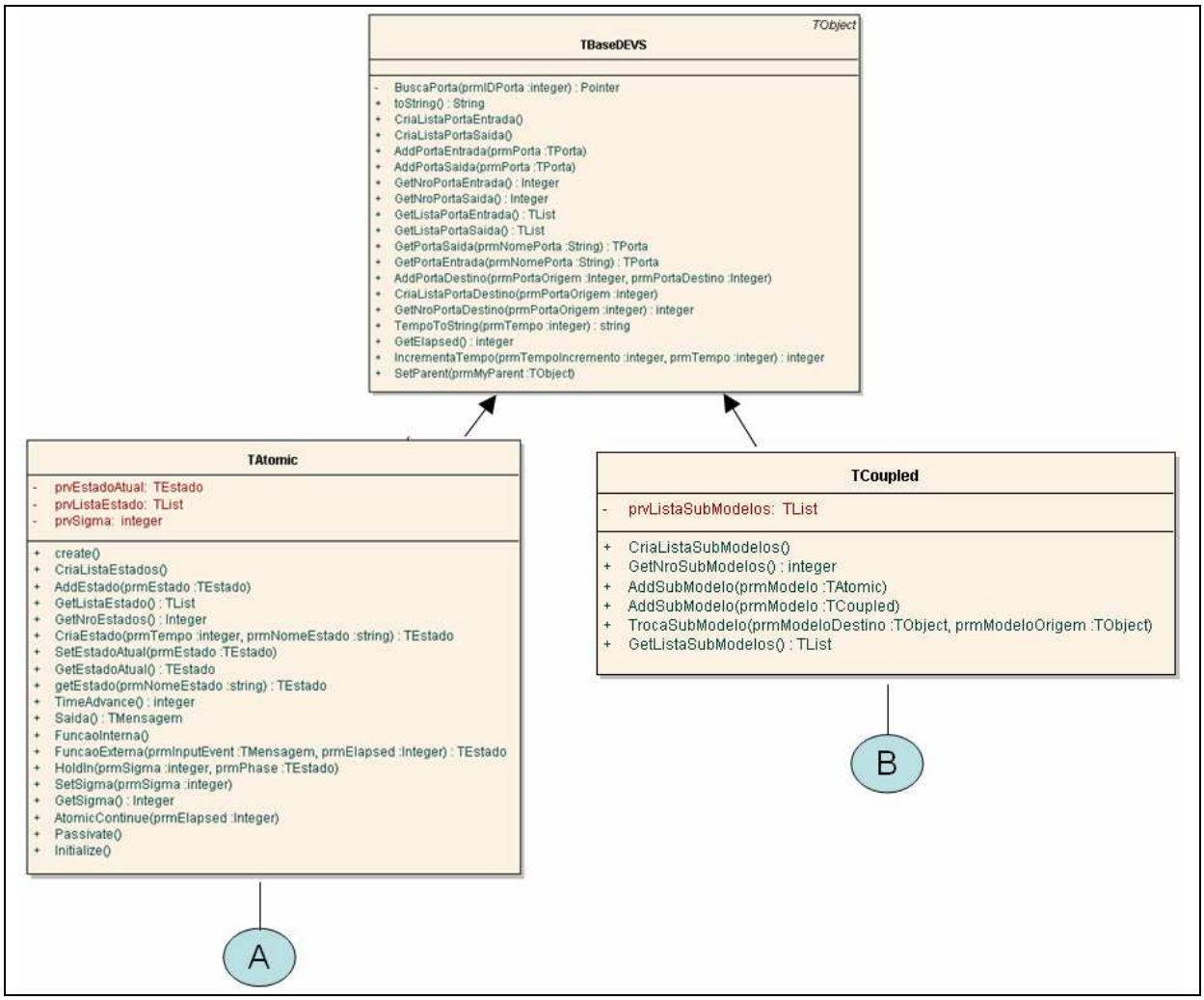


Figura 30 - Diagrama de classes detalhado

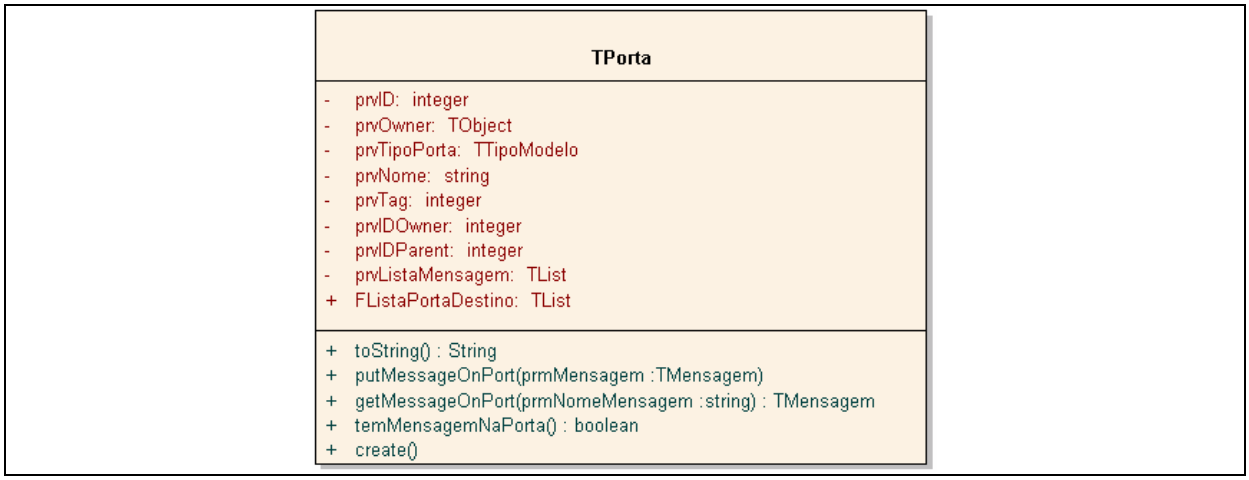


Figura 31 - Diagrama de classes detalhado

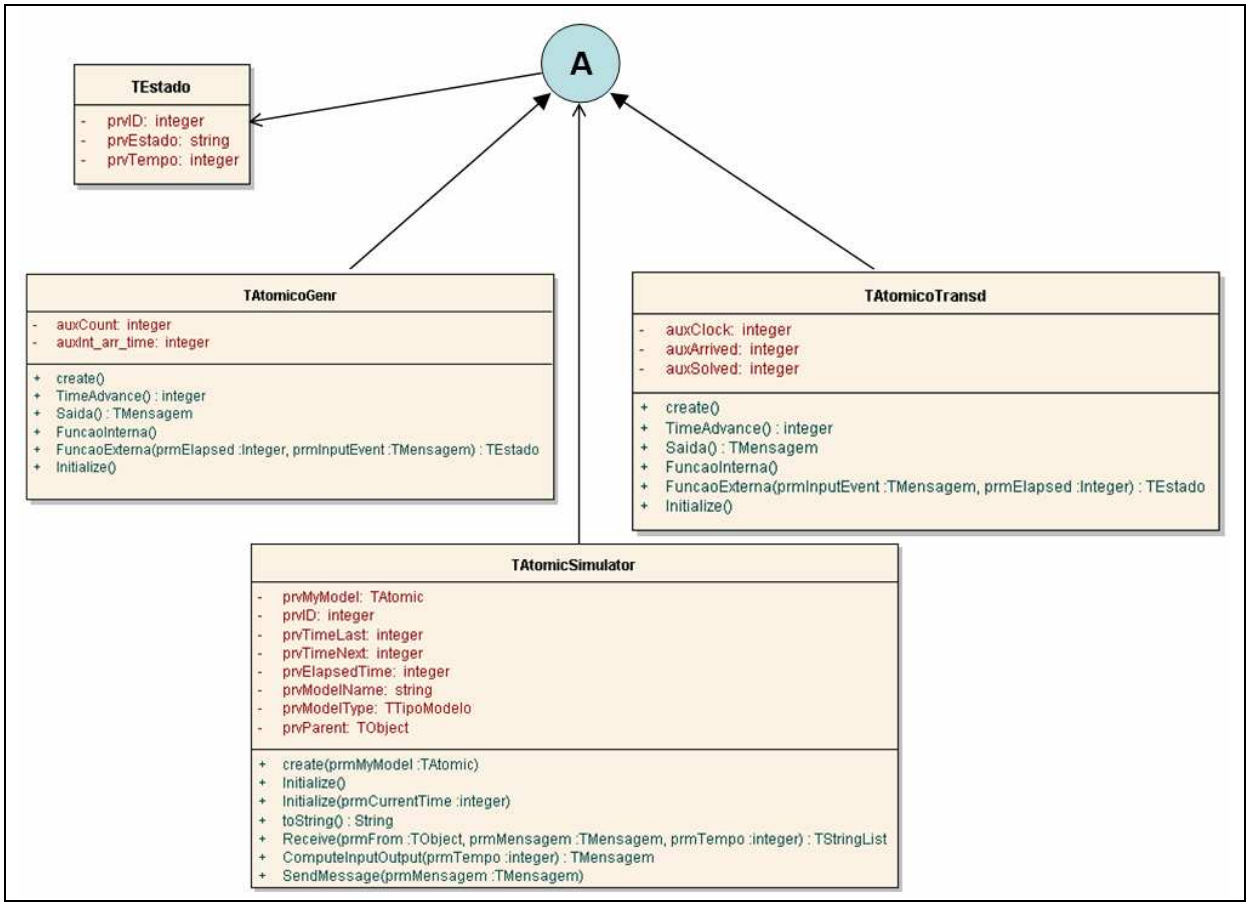


Figura 32 - Diagrama de classes detalhado

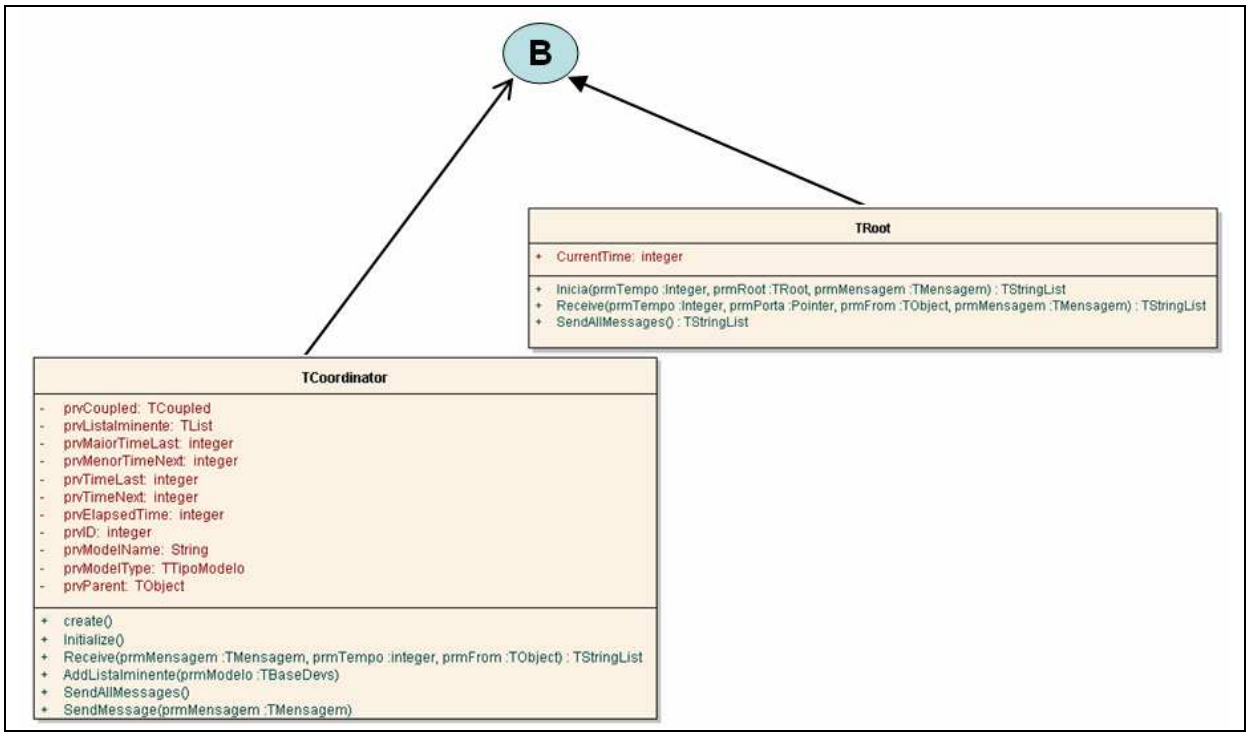


Figura 33 - Diagrama de classes detalhado