UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE CIÊNCIA E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

SUPORTE À ANÁLISE E VERIFICAÇÃO DE MODELOS RPOO

JOSÉ AMANCIO MACEDO SANTOS

CAMPINA GRANDE – PB

FEVEREIRO DE 2003

Suporte à Análise e Verificação de Modelos RPOO

José Amancio Macedo Santos

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Jorge César Abrantes de Figueiredo

(Orientador)

Dalton Dario Serey Guerrero

(Orientador)

Campina Grande, Paraíba, Brasil

SANTOS, José Amancio Macedo

S237S

Suporte à Análise e Verificação de Modelos RPOO

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, PB, Fevereiroo de 2003.

136p. Il.

Orientadores: Jorge César Abrantes de Figueiredo

Dalton Dario Serey Guerrero

1. Engenharia de Software
2. Redes de Petri
3. Orientação a Objetos

CDU – 519.711

Resumo

Redes de Petri Orientadas a Objetos (RPOO) é um formalismo que integra conceitos de redes de Petri e orientação a objetos. A estratégia adotada para a integração garante a manutenção das características originais de cada um dos paradigmas. Este é um aspecto importante para o aproveitamento dos recursos obtidos com o uso das redes de Petri e da orientação a objetos. Devido às suas características, RPOO é adequado para a especificação e verificação de sistemas distribuídos de software. Contudo, embora o formalismo esteja plenamente desenvolvido, a falta de ferramentas de suporte inviabiliza o uso prático de RPOO.

O problema central tratado neste trabalho é a falta de condições adequadas para a aplicação de RPOO. Mais especificamente, a falta de ferramentas de suporte para a validação e verificação de modelos. Abordamos o problema efetuando uma revisão do formalismo e construindo um sistema para a simulação de modelos. Como resultado da revisão, definimos uma linguagem de modelagem que concretiza o formalismo em termos computacionais. A definição desta linguagem, por sua vez, viabiliza a construção de ferramentas de simulação para os modelos, facilitando sua análise e aproximando o formalismo do uso prático.

Abstract

Oriented-Object Petri Nets (RPOO) is a formalism that integrates Petri Nets and Object-Oriented concepts. This composited strategy guarantees the preservation of the original characteristics of them both. This is an important aspect to take advantage of Petri Nets and Object-Oriented ideas. RPOO is appropriate to distributed systems verification and specification. Nevertheless, the absence of software tools makes the practical use of RPOO difficult.

In this work, the main problem we tackle is the absence of the appropriate conditions to the practical use of RPOO. For this reason, a revision of the RPOO formalism was done and a simulation system of RPOO models was built. On the revision, we defined a concret language in order to allow the implementation of support tools. The language is the base on wich the RPOO model simulation system was built.

Agradecimentos

Aos meus pais, José Roosevelt e Araiza, pelo apoio em todos os meus projetos pessoais e pelo carinho, eterno e incondicional. Também tenho um agradecimento especial a fazer para meu irmão Germano (Bicudinho) pela força e amizade em todos os momentos.

Um agradecimento também muito especial à maior incentivadora para o meu ingresso no mestrado, minha professora Rita Suzana. Também pelo incentivo, agradeço ao colega Eduardo Jorge.

Aos professores Jorge Abrantes e Dalton Serey, pelos ensinamentos e orientação, fundamentais para a conclusão do trabalho. Ao pessoal da iniciação científica, Paulo Barbosa, Taciano Silva e Rodrigo Tavares pela esforço e contribuição. Neste sentido, agradeço também aos colegas Cássio Leonardo e Emerson Ferreira. E a Érica de Lima Gallindo, sempre pronta para ajudar em qualquer momento.

A todos que tiveram uma participação direta ou indireta em minha vida durante o período de construção deste trabalho, mas que não foram mencionados: família, amigos, funcionários da universidade e colegas de curso.

Conteúdo

Lista de Figuras

Lista de Tabelas

Bibliografia

[BBG01] O. Biberstein, D. Buchs, and N. Guelfi. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 formalism. In F.DeCindio, G.A.Agha, and G.Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, Berlin, 2001.

[BCC01] E. Battiston, A. Chizzoni, and F. De Cindio. CLOWN as a Testbed for Concurrent Object-Oriented Concepts. In F.DeCindio, G.A.Agha, and G.Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, Berlin, 2001.

[BdC88] E. Battiston and F. de Cindio. OBJSA Nets: A Class of High-level Nets having Objects as Domains Advances in Petri Nets. In G. Rozenberg, editor, *Lecture Notes in Computer Science 340*. SpringerVerlag, 1988.

[Bjo85] Dines Bjorner. The raise project: Fundamental issues and requeriments. Technical report, RAISE/DCC/EM/1, Lingby, December 1985.

[Can] E. D. Canedo. Modelagem do Protocolo OSPF - Open Shorted Path First. Comunicação Interna.

[Can02] Edna Dias Canedo. Estudo e validação de uma linguagem de modelagem de sistemas baseada em redes de petri e orientação a objetos. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, 2002.

[des] *Manual Design-CPN*. Página referência da ferramenta Design-CPN, http://www.daimi.au.dk/designCPN/man/.

[DR98] J. Desel and W. Reisig. Place/Transition Petri Nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Net I: Basic Models*, volume 1491 of *Advances in Petri Nets*. Springer, Berlin, 1998.

[DSGP01] J. C. A. Figueiredo D. S. Guerrero and A. Perkusich. An Object-Based Modular CPN Approach: Its Application to the Specification of a Cooperative Editing Environment. In F.DeCindio, G.A.Agha, and G.Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, Berlin, 2001.

[ECG02] J. C. de Figueiredo E. Canedo, J. A. Santos and D. Guerrero. Experimenting a notation based on petri nets and object-oriented concepts. *I Brazilian Petri-Nets Meeting*, 2002.

[ECP98] Orna Grumber Edmund Clarke and Doron Peled. *Model Checking*. MIT Press, 1998.

[GL81] H. J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science 13*, pages 109–136, 1981.

[GPdF01] E. L. Galindo, A. Perkusich, and J. C. A. de Figueiredo. Aplicação de uma Notação baseada em Redes de Petri e Orientação a Objetos: Um Experimento de Modelagem. In *IV Workshop de Métodos Formais*, Rio de Janeiro - Brasil, Outubro 2001.

[Gue97] D. S. Guerrero. Sistemas de Redes de Petri Modulares Baseadas em Objetos. Dissertação de mestrado, UFPB - COPIN, 1997.

[Gue98] D. S. Guerrero. Orientação a objetos e modelos de redes de petri. Technical report, UFPB - COPELE, Abril 1998.

[Gue02] Dalton Dario Serey Guerrero. Redes de petri orientadas a objetos. Tese de doutorado, COPELE - Universidade Federal da Paraíba, 2002.

[IJB99] J. Rumbaugh I. Jacobson and G. Booch. *The Unified Modeling Language*. Addison-Wesley, 1999.

[Jen92a] K. Jensen. *Coloured Petri Nets 1: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, Berlin, Alemanha, 1992.

[Jen92b] K. Jensen. *Coloured Petri Nets 2: Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer-Verlag, Berlin, Alemanha, 1992.

[JR91] K. Jensen and G. Rozenberg. *High-Level Petri Nets Theory and Application*. Springer-Verlag, 1991.

[jun] *JUnit*. Página referência do JUnit, http://www.junit.org/.

[Lak01] C. Lakos. Object Oriented Modeling with Object Petri Nets. In F.DeCindio, G.A.Agha, and G.Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, Berlin, 2001.

[Lar98] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 1998.

[Lil01] J. Lilius. OB(PN) 2 : An Object Based Petri Net Programming Notation. In F.DeCindio, G.A.Agha, and G.Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, Berlin, 2001.

[LK94] C. A. Lakos and C. Keen. LOOPN++: A New Language for Object-Oriented Petri Nets. In *Proc. of Modelling and Simulation(European Simulation Multiconference)*, Barcelona, 1994.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mur89] T. Murata. Petri Nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[Pau96] Lawrence C. Paulson. *ML for the Working Programmer*. University of Cambridge, 1996.

[pos] *Poseidon*. Página referência da ferramenta Poseidon, http://www.gentleware.com/products/index.php3.

[RE98] G. Rozenberg and J. Engelfriet. Elementary Net Systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Net I: Basic Models*, volume 1491 of *Advances in Petri Nets*. Springer, Berlin, 1998.

[Rod02] Cássio Leonardo Rodrigues. Verificação de modelos em redes de petri orientadas a objetos. Proposta de dissertação de mestrado, COPIN - Universidade Federal de Campina Grande, 2002.

[ros] *Rational Rose*. Página referência da ferramenta Rational Rose, http://www.rational.com/products/rose/index.jsp.

[Sai96] Hossein Saiedian. An Invitation to Formal Methods. *Computer Society - IEEE*, 29(4):16–30, 1996.

[San01] J. A. Santos. Modelagem do Framework para Execução Distribuída em BETA - Uma proposta em RPOO e comparação com o modelo HCPN. Relatório da disciplina Projeto de Redes de Petri, oferecida pela COPIN- UFPB, Outubro 2001.

[Som96] I. Sommerville. *Software Engineering*. Addison-Wesley, 1996.

Apêndice A

Linguagem para Modelagem RPOO

Neste Apêndice, apresentamos a gramática para descrição dos modelos RPOO. Descrevemos a gramática explicando cada parte que representa os modelos durante a apresentação de cada item, seja classe, redes de Petri (contendo os domínios e operações) , inscrições de interação ou configuração inicial do sistema de objetos.

A.1 Gramática

A gramática para descrever modelos RPOO Simplificados tem o seguinte formato:

```
      <Modelo> ::= <Declaracoes> <Classes> <EstruturaInicial>
     <Classes> ::= <Classe> | <Classe> <Classes>
      <Classe> ::= class <IdClasse> { <Corpo> }
    <IdClasse> ::= <Palavra>
       <Corpo> ::= <Lugares> <Transicoes>
<Declaracoes> ::= Sigma = { <Cores> <Variaveis>
                            <Valores> <Funcoes> }
```

Modelo RPOO e Corpo das classes A variável inicial da gramática é `<Modelo>`. Um "modelo" RPOO é formado por um conjunto de classes (definido na variável `<Classes>`), uma configuração inicial e um conjunto de declarações.

Uma classe é identificada pela palavra reservada ***class*** seguida do nome da classe (variável <IdClasse>) e do corpo da classe, que é representado entre "chaves". A variável <IdClasse> é definida pela variável <Palavra>. Considere que, para efeito de notação, todas as palavras reservadas possuem o mesmo formato aplicado ao termo ***class***. No corpo das classes há uma "seção" para declaração dos elementos que podem ser manipulados nas redes (variáveis, funções , valores constantes e tipos de dados). Esta seção é identificada pela palavra reservada ***Sigma*** e as declarações devem estar entre os símbolos *{* e *}*. No restante do corpo das classes devem ser declarados os itens que compõem a estrutura da rede (lugares e transições).

A.1.1 Declaração de variáveis, tipos, funções e constantes

É possível descrever no corpo de *Sigma* um conjunto de elementos de quatro tipos diferentes, <Cor>, <Var>, <Val> e <Fun>, separados pelo símbolo ";". Estes elementos são cores, variáveis, constantes e funções definidas pelos modeladores. O símbolo ";" é utilizado também por algumas linguagens de programação, como Java por exemplo, para indicar o final de um comando. Antes de descrever cada um destes elementos, contudo, vamos apresentar a gramática que representa os domínios propostos para a linguagem:

<TipoGlobal> ::= <TipoElementar> | <TipoComposto>

<TipoElementar> ::= ***int*** | ***char*** | ***boolean*** | <Ref>

<Ref> ::= <IdClasse>

<TipoComposto> ::= ***string*** | <Enumerado> | <Produto> | <Lista>

Tipos de dados A variável <TipoGlobal> representa o conjunto formado pelos domínios da linguagem. A variável <TipoElementar> descreve os tipos elementares. A relação entre os nomes utilizados em <TipoElementar> e os domínios definidos é fácil de ser percebida, pois os identificadores usados na gramática são conhecidos e próximos dos tipos definidos. Assim, ***int***, ***char*** e ***boolean*** são sortes para os tipos de dados Inteiro, Char e Verdade, respectivamente. A sorte usada para definir um tipo <Ref>, que representa o domínio dos dados Ref, é o nome de alguma classe do modelo.

A variável <TipoComposto> definida na gramática descreve os tipos de dados com-

postos. A sorte ***string*** representa o domínio `Palavras`. As variáveis `<Enumerado>`, `<Produto>` e `<Lista>` descrevem os tipos `Enumerado`, `Produto` e `Lista`. Nas próximas definições apresentamos a sintaxe para descrição de cores e dos tipos compostos (enumerados, produtos e listas).

```
<Cores> ::= ∅ |<Cor> ; | <Cor> ; <Cores>
<Cor> ::= color <IdCor> = <TipoGlobal>
<IdCor> ::= <Palavra>
<Enumerado> ::= with <ListaPalavras>
<ListaPalavras> ::= <PalavraEnum> |
                    <PalavraEnum> "|" <ListaPalavras>
<PalavraEnum> ::= <Palavra>
<Produto> ::= product <ListaTipos>
<ListaTipos> ::= <Tipo> | <Tipo> * <ListaTipo>
<Tipo> ::= <TipoElementar> |
           <IdCor> |
           string
<Lista> ::= list <Tipo>
```

A variável `<Cor>` representa a definição de um domínio de dados que pode ser utilizado nas redes de Petri. Devemos utilizar a palavra reservada *color*, seguida de um nome dado pelo modelador para a cor (representada por `<IdCor>`). A descrição da cor utiliza "palavras" formadas dentro do alfabeto composto pelos caracteres definidos na linguagem. Logo após a descrição da cor, deve ser definido o domínio dos dados que ela pode assumir, que é um `<TipoGlobal>`, ou seja, ou é um tipo elementar, ou um dos tipos compostos definidos previamente. A descrição da cor e o domínio de dados a que ela pertence são separados pelo símbolo ":". Os tipos enumerados são definidos utilizando a palavra reservada ***with*** e uma lista de palavras separadas pelo símbolo "|". Uma cor formada por um tipo enumerado identifica o conjunto de valores constantes, ou descrições, que podem ser utilizados na rede. A variável `<Produto>` define um domínio que é formado por uma tupla contendo tipos de dados diferentes. Os tipos são separados pelo símbolo * e podem ser identificados com uma

das sortes para os tipos elementares, com a sorte ***string***, ou com a descrição de uma cor definida previamente (variável <Tipo> na gramática). A palavra reservada ***list*** indica a definição de uma cor do tipo lista. O tipo da lista também é definido pela variável <Tipo> na gramática .

Variáveis É possível declarar variáveis que são manipuladas nas redes de Petri para representar valores diferentes para fichas e expressões, ou seja, os diferentes modos de disparo das transições. Para definir uma variável utilizamos a seguinte sintaxe:

```
<Variaveis> ::= ∅ | <Var> ; | <Var> ; <Variaveis>
      <Var> ::= var <ListaIdVar> : <IdCor> [ = <Constante> ]
<ListaIdVar> ::= <IdVar> | <IdVar> , <ListaIdVar>
    <IdVar> ::= <Palavra>
```

A palavra reservada ***var*** indica que estamos declarando uma variável que recebe valores de acordo com o domínio definido para ela no modo de disparo de uma transição. A sintaxe requer a definição de um nome, seguido da cor (<IdCor>) que indica os possíveis valores que podem ser atribuídos à variável. Cada linha de declaração é encerrada pelo símbolo ";". O nome da variável e o tipo a que ela pertence são separados pelo símbolo ":". É possível definir um conjunto de variáveis em uma única linha de comando, como nas linguagens convencionais de programação, separando os identificadores pelo símbolo ",". A semântica de uma variável nesta linguagem é a convencional, ou seja, qualquer valor pode ser atribuído a ela (neste caso, no modo de disparo de uma transição), desde que seja do mesmo domínio de dados a que a variável pertence. Opcionalmente é permitido inicializar uma variável. Naturalmente, a forma de representação dos valores que inicializam as variáveis deve ser compatível com o tipo definido. Mais adiante discutimos a forma de representação para variáveis, definida na variável <Constante>. Definir um conjunto de variáveis em uma única linha de comando e utilizar a opção de inicialização implica em atribuir a todas as variáveis definidas o valor indicado.

Constantes As constantes são definidas utilizando a palavra reservada ***val*** seguida do nome da constante. Após o nome da constante devemos declarar o tipo de dado que ela

assume, definido a partir de uma cor (`<IdCor>`). Por fim, utilizamos o símbolo "=" seguido do valor efetivo que a constante deve assumir. No caso das constantes, uma vez definido um valor, este não pode ser alterado. O valor deve ser atribuído à constante no momento da declaração e deve ser descrito em um formato que possa ser interpretado dentro do domínio da constante. Por exemplo, se definimos que o domínio de uma constante pertence ao tipo ***int***, só podemos atribuir valores numéricos. Definiremos o formato da descrição para cada tipo mais adiante.

<Valores> ::= ∅ | <Val> ; | <Val> ; <Valores>

<Val> ::= ***val*** <IdVar> : <IdCor> = <Constante>

Funções Definimos agora a gramática para declaração de funções. Os usuários podem definir funções que permitem o tratamento de dados e operações sobre os tipos definidos.

```
<Funcoes> ::= ∅ | <Fun> ; | <Fun> ; <Funcoes>
<Fun> ::= fn <AssFuncao>
<AssFuncao> ::= <IdFun> ( <ListaArgumentos> )
                [ : <Tipo> ] =
                    <CorpoFuncao> |
                <IdFun> ( <ListaArgumentos> )
                [ : <Tipo> ] =
                    <CorpoFuncao> | <AssFuncao>
<IdFun> ::= <Palavra>
<ListaArgumentos> ::= ∅ |
                <Argumento> |
                <Argumento> , <ListaArgumentos>
<Argumento> ::= <IdArg> [ : <Tipo> ]
                <IdArg>::<IdArg> [ : <Tipo> ]
<IdArg> ::= <Palavra> | _
<CorpoFuncao> ::= <Constante> | <IdArg> | <Expressao>
```

Para declarar funções utilizamos a palavra reservada ***fn***, seguida de sua assinatura (`<AssFuncao>`). A assinatura tem o nome descrito pela variável `<IdFun>`, a lista de argumentos e o corpo da função. O tipo de retorno da função é opcional e deve ser declarado depois da lista de argumentos. A lista de argumentos pode conter um ou mais elementos, ou pode não conter nenhum elemento (símbolo ∅). Cada argumento tem um nome e, opcionalmente, um tipo. Se o tipo não for utilizado para o nome do argumento, este pode assumir qualquer valor, desde que as operações sejam válidas para os valores atribuídos. Assim, em chamadas diferentes uma função pode receber tipos de dados diferentes. Também na descrição de um argumento pode ser utilizado o símbolo _, que representa um valor nulo, ou

que deve ser desconsiderado. Este recurso é utilizado para a definição de funções por casos. Uma função pode ser declarada com várias cláusulas, separadas pelo símbolo `|`. Cada cláusula trata um tipo de padrão de argumentos. Esta é a forma de definir funções por casos adotada pela linguagem ML e aproveitada nas definições de CPN-ML. Como utilizamos o mesmo conjunto de elementos e significados, a semântica para a linguagem que apresentamos é a mesma de ML e CPN-ML e, portanto, daremos apenas uma breve explicação para a semântica adotada na definição de funções.

Como tratamos de uma linguagem com características funcionais, as funções "sempre" retorna algum valor. O corpo de uma função pode ser visto como expressões que retornam valores depois de avaliadas. Funções também podem retornar algum valor definido explicitamente (representado pela variável `<Constante>`) ou algum dos nos nomes dos argumentos definidos na função, já que estas descrições assumem valores quando a função é executada. Os valores resultantes da avaliação da função devem estar de acordo com o tipo de retorno definido, caso tenha sido explicitado. Se o tipo de retorno da função não foi definido, qualquer valor poderá ser retornado. Contudo, no caso de existir mais de uma cláusula, todas

devem retornar dados dentro do mesmo domínio.

```
<Expressao> ::= <Expressao> , <Expressao> |
                ( <Expressao> ) |
                [ <Expressao> ] |
                <ExpressaoNumerica> |
                <ExpressaoTeste> |
                <ExpressaoLogica> |
                <ExpressaoString> |
                <ExpressaoFuncao> |
                <ExpressaoIf> |
                <ExpressaoLista> |
                <ExpressaoProduto> |
                <Constante> |
                <IdArg> |
                <IdVar>
```

Expressões Uma expressão pode ser considerada como um tipo de informação que sempre resulta em um valor, depois de avaliada. Para as expressões definimos formatos de retorno, como `<Expressao>,<Expressao>`, *(* `<Expressao>` *)* (para descrever tuplas, representando produtos), ou *[* `<Expressao>` *]* (para descrever listas). Um valor declarado explicitamente (`<Constante>`), o nome de um argumento (`<IdArg>`), ou de uma variável (`<IdVar>`) também podem ser entendidos como expressões, pela possibilidade de serem avaliados diretamente. No caso de variáveis, contudo, elas não podem ser utilizadas no corpo de uma função. As expressões também representam operações que podem ser aplicadas a cada tipo de dado e que, após avaliadas, retornam o valor de um tipo específico. A seguir apresentamos a parte da gramática na linguagem para descrever esses casos.

```
<ExpressaoNumerica> ::= <Expressao>
                            ( + | - | * | / | %)
                        <Expressao>
   <ExpressaoTeste> ::= <Expressao>
                            ( > | < | >= |
                            <= | = | != )
                        <Expressao>
  <ExpressaoLogica> ::= not <Expressao> |
                        <Expressao>
                            ( and | or )
                        <Expressao> |
                        true |
                        false
  <ExpressaoString> ::= <Expressao> + <Expressao>
  <ExpressaoFuncao> ::= <IdFun>
                            ( <ListaChamada> )
      <ExpressaoIf> ::= if <Expressao> [ else <Expressao> ]
    <ListaChamada> ::= <ArgChamada> |
                        <ArgChamada>, <ListaChamada>
      <ArgChamada> ::= <IdArg> |
                        <Constante> |
                        null |
                        <Expressao>
   <ExpressaoLista> ::= <Expressao> @ <Expressao> |
                        <Expressao> :: <Expressao>
<ExpressaoProduto> ::= proj ( <Expressao> , <Expressao> )
```

A interpretação de cada uma das expressões é a convencional para operações com os tipos de dados definidos. Assim, daremos apenas uma breve descrição para cada uma delas.

- *Expressão Numérica* - as expressões numéricas retornam valores inteiros. O formato de uma expressão numérica é `<Expressao>` *operação* `<Expressao>`. Ambas expressões que compõem uma expressão numéricas podem ter qualquer formato, contanto que o resultado de sua avaliação seja um valor inteiro. Assim, expressões numéricas manipulam e retornam valores inteiros. As operações sobre os inteiros são as convencionais: adição, subtração, multiplicação, divisão e *mod*.

- *Expressão Teste* - para as expressões de teste, as operações aplicadas dependem dos tipos de dados tratados. A primeira consideração que devemos fazer é que ambas as expressões devem ser avaliadas para o mesmo domínio de dados. Em seguida devemos considerar qual o domínio está sendo tratado na expressão. Se a expressão efetua comparações sobre tipos inteiros, todas as operações (maior, menor, maior que, menos que, igualdade, não igualdade) definidas na sintaxe descrita na gramática podem ser consideradas em seu significado convencional. Para qualquer outro tipo de dado, as únicas operações válidas são a de igualdade e não igualdade. Para o caso de comparação entre dois produtos, cada elemento da uma tupla deve ser avaliado junto com o elemento correspondente da outra tupla que faz parte da expressão. A comparação para lista é feita de forma análoga, ou seja, comparando cada elemento de uma lista com o elemento da posição correspondente na outra lista que representa a expressão. Para os outros tipos, inclusive o tipo `<Ref>`, a semântica é a convencional.

- *Expressão Lógica* - as expressão lógicas manipulam e retornam valores *booleanos*. Os valores **`true`** e **`false`** também podem ser considerados expressões lógicas. Mais uma vez a semântica é a convencional para os operadores *not*, *and* e *or*.

- *Expressão String* - chamamos *expressão string* a manipulação dos valores *string* e caracter. Assim as expressões contidas em `<Expressao>` + `<Expressao>`, devem ser ou do tipo caracter ou do tipo *string*. A única operação permitida em `<ExpressãoString>` é a de concatenação (representada pelo símbolo "+") e retorna sempre uma *string*.

- *Expressão Função* - para este caso temos a chamada de alguma função definida previamente. Assim, qualquer tipo de dado pode ser manipulado, já que qualquer valor pode ser passado como argumento. Qualquer tipo definido pode ser retornado, dependendo das operações que a função efetua com os valores dos argumentos recebidos. Os valores passados como argumentos podem ser um dos argumentos recebidos, um valor definido explicitamente (`<Constante>`), o resultado da avaliação de uma outra expressão, ou o valor ***`null`***, que representa a ausência de valor.

- *Expressão Lista* - é possível concatenar listas ou extrair elementos de uma lista. A operação de concatenação (símbolo *@*) requer que a avaliação das duas expressões que formam a operação sejam listas de um mesmo tipo de dado. O símbolo *`::`* nos permite extrair elementos de uma lista. Vejamos o seu significado através de um exemplo. Suponha que temos em uma função um argumento *a::b*. Deste argumento podemos extrair o elemento *a*, que representa o primeiro elemento da lista. Assim, o retorno do elemento anterior a *`::`* é um tipo de dado qualquer. A parte final, ou seja, *b* em nosso exemplo, é o restante da lista e, portanto, do tipo lista. Podemos também compor da seguinte forma: *a::b::c*. Neste caso *a* e *b* são o primeiro e o segundo elemento da lista, respectivamente, e *c* é a lista restante. Da mesma forma que podemos extrair um elemento de uma lista é possível também inserir. Por exemplo, se temos um valor *a* de um determinado tipo, e uma lista *b* formada por elementos do mesmo tipo de *a*, então *a::b* insere o elemento *a* no início da lista e *b::a* insere o elemento *a* no final da lista *b*. Assim, o domínio dos valores da lista *b* deve ser do mesmo tipo do valor *a*.

- *Expressão Produto* - definimos uma função que permite extrair os elementos de um produto. A função ***`proj`*** (de *projeção*) possui dois argumentos, que podem ser expressões de qualquer tipo. Desde que o primeiro argumento seja do tipo `Produto` e o segundo argumento resulte em um valor `Inteiro`. A função ***`proj`*** deve retornar do produto cuja posição foi indicada pelo valor numérico passado no segundo argumento. A posição é indicada pela ordem definida na declaração do tipo.

Cada tipo de expressão mencionado manipula e retorna determinados tipos de dados. Descrevemos agora como estes dados são representados nesta linguagem. Estamos chamando

os valores finais que cada tipo de dado pode assumir de <Constante>.

<Constante> ::= <ConstanteNumerica> |
<ConstanteChar> |
<ConstanteString> |
<ConstanteLista> |
<ConstanteVerdade> |
<ConstanteEnum> |
<ConstanteProd> |
<ConstanteRef>

Símbolos dos domínios de dados Definimos uma variável <Constante> para cada domínio de dados identificados nos modelos. As constantes representam a forma como descrevemos os dados de cada domínio e, portanto, definimos uma variável

(<ConstanteNumerica>, <ConstanteNumerica>, etc.) para representar cada tipo.

```
<ConstanteNumerica> ::= 0 | 1 | 2 | 3 | 4 |
                        5 | 6 | 7 | 8 | 9 |
                        <ConstanteNumerica> • <ConstanteNumerica>
    <ConstanteChar> ::= ! | # | $ | % | & | | ( |
                        ) | - | + | = | < | > | : |
                        ; | . | , | ? | ^ |   | | |
                        a | b | c | d | e | f | g |
                        h | i | j | k | l | m | n |
                        o | p | q | r | s | t | u |
                        v | w | x | y | z | A | B |
                        C | D | E | F | G | H |
                        I | J | K | L | M | N | O |
                        P | Q | R | S | T | U | V |
                        W | X | Y | Z
   <ConstanteAscii> ::= <ConstanteNumerica> | <ConstanteChar>
  <ConstanteString> ::= "<ConstanteAscii>" |
                        "<ConstanteAscii> • <ConstanteString>"
   <ConstanteLista> ::= [ <Constante> ]
 <ConstanteVerdade> ::= true | false
    <ConstanteEnum> ::= <PalavraEnum>
    <ConstanteProd> ::= ( <ListaConstante> )
   <ListaConstante> ::= <Constante> | <Constante> , <ListaConstante>
     <ConstanteRef> ::= <Palavra> | this
          <Palavra> ::= <ConstanteString>
```

Comecemos nossa análise pelas constantes numéricas (definidas na variável <ConstanteNumerica>). Este tipo de constante representa o domínio dos inteiros. Cada

inteiro pode ser formado pelos símbolos `0` a `9` ou por qualquer composição destes elementos. O significado de cada símbolo formado será o valor inteiro que ele representa. Para a manipulação de números inteiros, seja em variáveis, valores constantes ou expressões, devemos utilizar o conjunto de elementos descritos pela variável `<ConstanteNumerica>`.

Na variável `<ConstanteChar>` representamos todos os caracteres que podem ser utilizados na linguagem. Basicamente temos os caracteres *ascii* como o conjunto de símbolos que formam as palavras que podem ser utilizadas nos modelos. Estas palavras são representadas pela variável `<ConstanteString>`, entre "aspas duplas". Para representar listas, descrevemos os elementos que fazem parte da lista entre colchetes ("[" e "]"). A variável `<ConstanteVerdade>` representa o domínio de dados *booleanos*. Apenas dois valores são aceitos, descritos pelas palavras reservadas **`true`** e **`false`**. O significado de cada palavra é o convencional. Os valores possíveis de serem assumidos pelos tipos enumerados são as palavras definidas na declaração de cores do tipo enumerado. Para o tipo produto, seus elementos são descritos entre parêntesis e separados por vírgula. E para o tipo `Ref` podemos utilizar qualquer palavra para representar o identificador de um objeto, ou a palavra reservada **`this`**. O termo **`this`** é um elemento da linguagem que pertence ao domínio `Ref` e que, ao ser avaliado, retorna o identificador do próprio objeto que o utiliza.

A.1.2 Sintaxe para definição de lugares e transições

Na seção anterior apresentamos a gramática para descrever variáveis, tipos, constantes e funções. Nesta seção descrevemos a gramática para representar os lugares e transições das redes. Relembre que a definição da variável `<Corpo>` é

$$\texttt{<Corpo>} ::= \texttt{<Lugares>}\,\texttt{<Transicoes>}$$

Uma seção especial foi dedicada à descrição das variáveis, funções, constantes e cores devido à quantidade de elementos que formam as variáveis definidas na gramática. Nesta seção vamos apresentar a sintaxe para definição do restante do corpo de uma classe. Ou seja, vamos definir a gramática para representar lugares e transições das redes que descrevem o comportamento de uma classe. Temos a seguinte gramática:

```
<Lugares> ::= ∅ | <Lugar> ; | <Lugar> ; <Lugares>
  <Lugar> ::= Place <ListaIdLugar> : <IdCor>
                [ =<Marcacao> ]
<ListaIdLugar> ::= <IdLugar> | <IdLugar> , <ListaIdLugar>
  <IdLugar> ::= <Palavra>
 <Marcacao> ::= [ <ConstanteNumerica> ` ] <Constante> |
                [ <ConstanteNumerica> ` ] <Constante> +
                <Marcacao>
```

Lugares da Rede Para descrever os lugares de uma rede, utilizamos a palavra reservada **Place** no corpo da classe. Cada comando para descrever lugares é separado pelo símbolo ";". É permitido declarar um conjunto de lugares com um mesmo domínio de cores, separando os identificadores dos lugares pelo símbolo ",". Após a palavra reservada **Place** deve-se indicar o(s) nome(s) e a cor que define o domínio de dados que podem ser armazenados nos lugares. O nome dos lugares é identificado pela variável <ListaIdLugar> e a cor pela variável <IdCor>. Estes elementos são separados pelo símbolo ":". Os lugares são identificados por uma <Palavra> e todos os lugares de uma rede devem ter nomes (<IdLugar>) diferentes.

Os lugares podem ser inicializados explicitamente e o formato é descrito na variável <Marcacao>. Lugares que não são inicializados explicitamente não possuem nenhuma ficha na marcação inicial da rede. Para definir a marcação de um lugar utilizamos a variável <ConstanteNumerica>, que representa a descrição de um valor inteiro. Neste caso os valores numéricos podem ser somente positivos. A <ConstanteNumerica> representa a quantidade de fichas do lugar e vem sucedida pelo símbolo ` e por uma <Constante> que representa a descrição do valor que vai ser armazenado inicialmente no lugar. Claro que o formato do valor descrito nos elementos representados sobre a variável <Constante> deve estar de acordo com o domínio definido para o lugar. A ausência da indicação da quantidade de fichas pertencentes a um lugar inicialmente indica a determinação

de apenas uma ficha no formato descrito. Para acrescentar outras fichas usamos o símbolo `+`.

Transições Descrevemos agora a variável `<Transicoes>` sintaticamente na gramática abaixo.

```
<Transicoes> ::= ∅ | <Transicao> | <Transicao> <Transicoes>
 <Transicao> ::= Transition <IdTrans> { <CorpoTrans> }
   <IdTrans> ::= <Palavra>
<CorpoTrans> ::= <Variaveis> <Funcoes> <Valores>
                 <PreCondicoes> <PosCondicoes>
                 [ <Comentarios> ]
```

Em uma analogia com as linguagens de programação convencionais, podemos visualizar o corpo de uma transição como o programa que descreve as ações de um método. Entretanto, como estamos lidando com um formalismo para descrição do comportamento de um objeto (que são as redes de Petri), a representação do efeito de uma transição não é uma seqüencia de ações, como em uma linguagem de programação convencional. O efeito de uma transição é representado como um conjunto de pré e pós-condições que indicam quando a transição pode disparar e qual o novo estado da rede após o disparo. O corpo da transição (variável `<CorpoTrans>`) é definido pelas variáveis `<PreCondicoes>` e `<PosCondicoes>`. As variáveis `<Variaveis>`, `<Funcoes>` e `<Valores>` já foram descritas, e são utilizadas aqui para permitir a definição de variáveis, valores constantes e funções locais às transições. A gramática a seguir descreve o formato para descrição das pré-condições e do

efeito do disparo das transições.

```
<PreCondicoes> ::= Pre { <CorpoPre> }
<PosCondicoes> ::= Pos { <CorpoPos> }
    <CorpoPre> ::= <Guarda> ; |
                   <Guarda> ; <ListaPreCond>
                   <ListaPreCond>
      <Guarda> ::= <Expressao>
<ListaPreCond> ::= <PreCond> ; |
                   <PreCond> ; <ListaPreCond>
     <PreCond> ::= <IdLugar> = <Marcacao>
    <CorpoPos> ::= <ListaPosCond> |
                   <ListaPosCond> <ListaInsc> |
                   <ListaInsc>
<ListaPosCond> ::= <PosCond> ; |
                   <PosCond> ; <ListaPosCond>
     <PosCond> ::= <IdLugar> = <Marcacao>
```

Para descrever as pré e pós-condições referentes ao disparo de uma transição, definimos uma "seção" para cada um dos itens. As palavras reservadas **`Pre`** e **`Pos`** identificam estas seções. As pré e pós-condições devem ser definidas entre chaves (variáveis `<CorpoPre>` e `<CorpoPos>`).

Na seção de *pré-condições* definimos a guarda e a quantidade de fichas retirada de cada lugar de entrada da transição quando do disparo da mesma. De acordo com o formalismo das redes de Petri, o disparo de uma transição acontece somente se a expressão da guarda (caso exista alguma) for avaliada para *true* e se existirem mais fichas no lugares de entrada da transição que as avaliadas em cada um dos arcos de entrada. A guarda é representada na variável `<Guarda>` e é uma expressão que deve ser avaliada sempre para *true* ou *false*. Para representar a condição de disparo relacionada a quantidade de fichas em um lugar de entrada da transição (variável `<PreCond>`) utilizamos o nome do lugar , seguido do símbolo "=" e de uma marcação. A interpretação neste caso é que o lugar indicado terá retirada

a quantidade de fichas avaliada na marcação. A guarda e cada uma das marcações que indicam a quantidade de fichas retiradas de um lugar são separadas pelo símbolo ";". Essa forma de representação torna implícita a descrição dos arcos da rede. Cada um dos lugares indicados na lista de pré-condições é um dos lugares de entrada da transição em questão. A variável <Marcacao> representa as expressões dos arcos que ligam os lugares à transição. Desse modo, o formato textual para descrição de modelos RPOO fica mais enxuto, pois não é necessário descrever os arcos explicitamente. Além disso, o fato dos elementos serem representados isoladamente e separados por ";" aproxima texto do formato utilizado nas linguagens de programação convencionais, se considerarmos que os elementos descritos no corpo das pré-condições representam os comandos que descrevem as condições para execução de um método.

Para descrever a "seção" *pós-condição* utilizamos o mesmo formato da seção pré-condições (observe que a variável <PreCond> é igual a <PosCond>). Ou seja, o nome do lugar, seguido do símbolo "=" e a marcação. Porém, a semântica é diferente. Neste caso, são identificados os lugares de saída e a marcação representa as fichas que serão inseridas no lugar, depois de avaliadas as expressões. Ou seja, a marcação representa as expressões dos arcos de saída das transições. Dessa forma representamos o efeito do disparo de uma transição. Também são descritas nesta "seção" as inscrições de interação (variável <ListstaInsc>). Todos os elementos são separados pelo símbolo ";" para seguir o mesmo padrão das linguagens de programação convencionais. A seguir descrevemos a forma de representar as

inscrições de interação.

```
<ListaInc> ::= ∅ | <Inscricao> ; |
                  <Inscricao> ; <ListaInsc>
<Inscricao> ::= <IdVar> . <Termo> |
                <IdVar> ! <Termo> |
                <IdVar> ? <Termo> |
                [ <IdVar> = ] new <IdClasse> |
                <IdVar> - |
                end
<Termo> ::= <RotuloMens> ( <ListaValores> )
<RotuloMens> ::= <PalavraEnum>
<ListaValores> ::= <Valor> | <Valor> , <ListaValor>
<Valor> ::= <IdVar> | <Expressao> | <Constante>
```

Para as inscrições de interação devemos utilizar alguma variável definida anteriormente (em `<IdVar>`). A variável deve ser do tipo `<Ref>`, ou seja, deve ser uma referência para o identificador de alguma classe do modelo. A variável `<Termo>` representa as mensagens enviadas, ou recebidas, entre os objetos e possui um rótulo e uma lista de valores passados como argumento. Os valores devem estar entre entre parêntesis e separados por `,`. O rótulo deve ser descrito por um valor definido em algum tipo enumerado. Com relação aos valores trocados, podemos ter variáveis, constantes, ou mesmo expressões. Para efeito da aplicação das ações de troca de mensagens sobre o sistema de objetos, o termo enviado vai conter o resultado da avaliação das variáveis ou expressões. A avaliação dos valores é considerada no modo de disparo da transição.

Neste ponto, temos uma observação sobre a inscrição de interação que representa a instanciação de objetos, descrita no formato

$$[\ \texttt{<IdVar>} =]\ \textbf{\textit{new}}\ \texttt{<IdClasse>}.$$

Para inscrições deste tipo temos a seguinte interpretação: uma nova instância do tipo da classe identificada em `<IdClasse>` é criada. Assim, um novo identificador deve ser gerado. O valor deste identificador deve ser armazenado na variável descrita por `<IdVar>`.

Consideramos que o corpo da classe em questão passa a descrever o comportamento do objeto instanciado. Descrevemos aqui o significado deste tipo de inscrição no modelo, mas é importante lembrar que essa é uma avaliação comportamental que tem efeito apenas sobre o sistema de objetos.

Outras Observações Definimos aqui algumas restrições quanto ao conjunto de identificadores elaborados. A primeira consideração é relacionada ao conjunto de nomes de classes (`<IdClasse>`). O conjunto formado pelos nomes das classes dos modelos não pode possuir elementos repetidos, ou seja, os nomes das classes devem ser únicos nos modelos. Com relação aos elementos criados dentro de uma classe, se considerarmos que {`<IdCor>`} é o conjunto formado pelos nomes das cores definidas nos modelos, {`<IdVar>`} é o conjunto formado pelos nomes das variáveis definidas nos modelos, e assim para todos os elementos, temos:

$$\{\texttt{<IdClasse>}\} \cap \{\texttt{<IdCor>}\} \cap \{\texttt{<PalavraEnum>}\} \cap$$
$$\{\texttt{<IdVar>}\} \cap \{\texttt{<IdFun>}\} \cap \{\texttt{<IdArg>}\} \cap$$
$$\{\texttt{<IdLugar>}\} \cap \{\texttt{<IdTrans>}\} = \emptyset$$

Além disso, cada um dos conjuntos descritos acima também não devem possuir elementos repetidos, ou seja, os nomes de lugares, de cores, etc. devem ser únicos dentro de uma classe.

Considerando o escopo dentro do corpo de uma transição, vale a regra:

$$\{\texttt{<IdVar>}\} \cap \{\texttt{<IdFun>}\} \cap \{\texttt{<IdArg>}\} = \emptyset$$

Pelas definições acima é possível declarar variáveis, ou mesmo funções com mesmo nome em escopos diferentes: para as classes ou para as transições. Para efeito de modelos, estes são elementos diferentes e na avaliação do disparo de uma transição, ou na definição de um modo de disparo, devemos considerar primeiro as variáveis ou funções definidas no corpo da transição, para depois avaliar as definições inseridas no escopo da classe.

A.1.3 Sistema de Objetos

Uma vez apresentado o formato para descrição dos elementos que compõem as classes dos modelos, precisamos definir uma sintaxe para a representação da estrutura inicial do sis-

tema de objetos. Relembre que temos a seguinte definição para os modelos RPOO nesta linguagem:

```
<Modelo> ::= <Declaracoes><Classes><EstruturaInicial>
```

Nas seções anteriores definimos uma gramática para representar as classes do modelo. Apresentamos agora a gramática para descrição da configuração inicial do modelo.

```
<EstruturaInicial> ::= InitialConf =
                       { <ListaIdentObjetos><DefListaObj> }
<ListaIdentObjetos> ::= <IdentObjetos> |
                        <IdentObjetos><ListaIdentObjetos>
<IdentObjetos> ::= <IdClasse><ListaId> ;
<ListaId> ::= <IdObj> |
              <IdObj> , <ListaId>
<DefListaObj> ::= Structure = <ListaObj>
                              [ + <ListaMens> ]
<ListaObj> ::= ∅ |
               <IdObj> [<Ligacoes> ] |
               <IdObj> [<Ligacoes> ] +
               <ListaObj>
<IdObj> ::= <Palavra>
<Ligacoes> ::= "[" <ListaLig> "]"
<ListaLig> ::= <IdObj> |<IdObj> , <ListaLig>
<ListaMens> ::= <Conteudo> ( <IdObj> , <IdObj> ) +
                <ListaMens>
<Conteudo> ::= <PalavraEnum> | < <IdObj> >
```

A estrutura inicial é formada por uma lista de objetos e, opcionalmente, por uma lista de mensagens. Usamos o símbolo + para acrescentar outros objetos à lista. Se um objeto possui ligações, estas devem se concatenadas ao objeto (entre colchetes). Os identificadores

dos objetos devem seguir a regra apresentada no início desta seção. As regra define que os identificadores dos objetos devem ser representados pelo menor conjunto de caracteres que identifica a classe mais um valor inteiro.

Cada mensagem da lista de mensagens deve ser representada pela descrição do seu conteúdo, que deve ser um elemento definido em algum tipo enumerado do modelo, ou uma referência (`<IdObj>`) para outro objeto. Caso o conteúdo seja uma referência deve ser inserido entre os símbolos `< >`. Entre parêntesis, depois do conteúdo da mensagem, devemos indicar os objetos origem e destino. As mensagens também são concatenadas entre si e à lista de objetos com o símbolo `+`.

Apêndice B

Artefatos do Projeto da Ferramenta SSO

Neste Apêndice, apresentamos o conjunto de artefatos produzidos no projeto para desenvolvimento do SSO. Apresentamos o manual da ferramenta, o diagrama de classes contendo as exceções tratadas e os diagramas de seqüência do SSO.

B.1 Manual do SSO

B.1.1 Introdução

Este manual foi desenvolvido para permitir maior facilidade na execução e utilização do Simulador de Sistema de Objetos (SSO). O SSO é uma ferramenta que permite simular as alterações ocorridas em um sistema de objetos pela execução de ações/eventos, de acordo com as regras introduzidas na formalização.

B.1.2 Instalação

Nesta seção é descrito o conjunto de arquivos que compõe o SSO. Para a execução do programa é necessário copiar os arquivos para um diretório e executar os "scripts"disponibilizados.

Conteúdo do pacote do Simulador de Sistemas de Objeto:

- *sisobj.jar* - arquivos fontes e *.class* do SSO.
- *docs.zip* - Arquivos do *javadoc* e o projeto do SSO (no formato *mdl*).

- *junit.jar* - pacote de testes de unidade necessário para rodar os testes do SSO.
- *iucsisobj* - script para executar a interface a caracter.
- *iugsisobj* - script para executar a interface gráfica.
- *testsisobj* - script para executar os testes de unidade.

Para executar os programas basta digitar o nome do "script"correspondente ou ./nome:

ex:

iucsisobj<enter>

./iucsisobj <enter>

- Conteúdo dos "scripts":

  iucsisobj - java -classpath .:sisobj.jar sisobj.vista.SistemaObjetos

  iugsisobj - java -classpath .:sisobj.jar sisobj.iugrafica.AppSistemaObjetos

  testsisobj - java -classpath .:sisobj.jar:junit.jar sisobj.testes.TesteControle

B.1.3 Simulação

O Simulador de Sistemas de Objetos dispõe de duas interfaces, uma a caracter e outra gráfica. Esta seção descreve como utilizar cada uma das interfaces e como deve acontecer a entrada de dados em cada uma delas.

Independente da interface utilizada, o usuário do SSO pode optar por duas formas distintas de inserção de dados: a partir de um arquivo ou com entrada iterativa da estrutura e eventos do sistema. O usuário pode, também, simular a execução do SSO com a entrada via arquivo e em seguida simular novas ações e/ou eventos de froma iterativa.

A execução do SSO deve ser feita utilizando a linguagem proposta, de acordo com a seção B.1.4

Simulando na Interface a Caracter

Para simular a execução do SSO os seguintes comandos devem ser executados.

- Entrada de dados via arquivo

  Para proceder corretamente basta digitar o script iucsisobj + o nome do arquivo.

  iucsisobj <nomeDoArquivo >

- Entrada de dados iterativa

  Para a entrada de dados iterativamente deve ser utilizado o "script"*iucsisobj*.

  Há um conjunto de funcionalidades oferecidos pelo SSO iterativo:

COMANDO	FUNCIONALIDADE
-i	imprime na tela a estrutura atual
-g$<nome\_arquivo>$	grava a estrutura atual em <nome_arquivo>
-h	imprime na tela o help do sistema
-s	sai do sistema ou Ctrl+d(Final do Arquivo)
-e	abre o arquivo *erros.log* no gvim (modo gráfico)
-si	abre o arquivo *simulação.log* no gvim (modo gráfico)
-r	reinicia o sistema (Apaga a estrutura atual)

Simulando na Interface Gráfica

A interface gráfica dispõe de quatro botões:

Para ler a estrutura de um arquivo deve-se digitar o nome do arquivo a ser lido no campo abaixo do nome *Ler do Arquivo* e, em seguida, clicar no botão *Executar*. Dessa maneira aparecerá a configuração inicial, presente no arquivo, na interface gráfica.

Os eventos registrados no arquivo são exibidos para que possamos acompanhar qual evento que levou de uma estrutura para outra.

Caso o usuário deseje ver as estruturas posteriores basta clicar no botão *Próximo*, dessa maneira irá executar as possíveis ações e eventos presentes no arquivo. Caso o usuário deseje visualizar as estrutura anteriores basta clicar no botão *Anterior*.

Ler do Teclado

Para executar o SSO a partir do teclado , basta colocar a estrutura desejada no espaço abaixo do nome *Ler do Teclado* e em seguida clicar em *Executar*, dessa maneira o usuário poderá ver desenhada a estrutura digitada. Para executar ações e eventos a partir do teclado o procedimento é análogo.

Posição Objeto

Para posicionar os objetos o usuário pode usar o botão da interface, onde colocará o rótulo do objeto e sua respectiva posição no eixos X e Y, e depois clicar em *Posicionar*.

Posição Mensagem

Para posicionar a mensagem usando o botão da interface é preciso apenas indicar qual a mensagem, seu destino e sua origem, e sua nova posição nos eixos X e Y, depois clicar em *Posicionar*.

- Posicionando com o mouse

  O mouse também pode ser utilizado para reposicionar objetos e mensagens. O procedimento deve ser o seguinte:

  – Primeiro Passo:

    Clique com o botão direito do*mouse* no objeto ou mensagem que se deseja mover. Nesse instante o objeto deve ficar vermelho.

  – Segundo Passo:

    Clique com o *mouse* na posição em que deseja colocar o objeto ou mensagem.

B.1.4 Entrada de Dados

Para simplificar a utilização do SSO, foram definidas uma gramática e uma linguagem para a entrada de dados no sistema, que utilizam a representação algébrica proposta em RPOO.

Para que seja iniciada a execução, o SSO deve receber uma estrutura que representa a configuração inicial do modelo RPOO. Em caso do SSO receber um arquivo de texto, a primeira linha de código será interpretada como a configuração inicial. Se, ao invés de um arquivo de texto, a simulação for feita de forma iterativa, a primeira estrutura digitada será vista como a configuração inicial.

Para representar uma configuração inicial no SSO, deve-se utilizar uma expressão algébrica contendo os elementos que fazem parte da configuração. Uma expressão algébrica é representada como uma estrutura e seus elementos devem ser compostos através do sinal indicativo de adição (+). Os elementos de uma estrutura são representados da seguinte forma:

- Objetos

  Um objeto é representado com um rótulo simples:

  Ex.: ‘a’ é o rótulo do objeto ‘a’.

- Ligações

Uma ligação é criada com o rótulo do objeto 'origem', e entre colchetes, o rótulo do objeto destino. Caso seja mais de um objeto destino, os rótulos devem ser postos entre colchetes separados por vírgula.

origem[destino1,destino2,destino3,....]

- Mensagens

  No SSO podemos ter dois tipos de mensagens. Pode-se enviar *dados* ou *referências* para outros objetos contidos na estrutura.

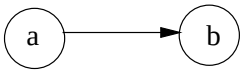
  Para representar uma mensagem contendo um dado, deve-se digitar o string desejado, e, entre parênteses, a origem da e o destino da mensagem separado por vírgula.

  msg(origem,destino)

  Para que um objeto envie uma mensagem contendo uma referência, deve-se colocar entre os sinais indicativos de "maior que"(<) e "menor que "(>), a referência que se deseja passar e, entre parênteses, a origem e o destino separado por vírgulas.

  <msg>(origem,destino)

- Exemplo de uma Possível configuração Inicial:a[b]+b



Nessa configuração inicial estão sendo criados os objetos

'a' ,'b' e uma ligação do objeto 'a' para o objeto 'b'.(Os símbolos serão

explicados adiante.)

Representação de Ações/Eventos

A execução de ações que é que altera efetivamente a estrutura. Para executar uma ação deve-se indicar o objeto que irá executar a ação seguido de dois pontos e o tipo da ação que ele irá desenvolver.

Veja a tabela comparativa:

AÇÃO	NOME	SSO
τ	ação local (ou interna)	$\#$
$newx$	criação ou instanciação de objetos	$+x$
$x?m$	entrada de dados	$x?m$
$x.m$	saída assíncrona de dados	$x.m$
$x!m$	saída síncrona de dados	$x!m$
$\widetilde{x}$	desligamento ou remoção de ligação	$-x$
end	ação final (ou auto-destruição)	$\sim$

As ações podem ser compostas em um único evento com o símbolo '@'.

*Exemplo:*a:b.m @ b:a!<c> @ a:b<c> @ b:#

- Exemplo de uma Ação Interna

  a:# (O objeto 'a' realizando uma ação interna)

  b:# (O objeto 'b' realizando uma ação interna)

- Exemplo de uma Ação de Criação

  a:+b (O objeto 'a' cria o objeto 'b')

  b:+c (O objeto 'b' cria o objeto 'c')

- Exemplo de uma Ação Entrada de Dados

  a:b?oi (O objeto 'a' consome a mensagem "oi" enviada por 'b')

  x:y?<z> (O objeto 'x' consome de 'y' uma referência para 'z')

- Exemplo de uma Ação Saída Assíncrona

  a:b.m (O objeto 'a' envia a mensagem "m" para o objeto 'b')

  a:b.<a> (O objeto 'a' envia para o objeto 'b' uma referência para ele próprio)

- Exemplo de uma Ação Saída Síncrona

  a:b!oi (O objeto ‘a’ executando uma saída síncrona para o objeto ‘b’ passando um mensagem “oi”.)

  a:b!<c> (O objeto ‘a’ executando uma saída síncrona para o objeto ‘b’ enviando uma referência para o objeto ‘c’.)

- Exemplo de uma Ação Final

  a:~ (O objeto ‘a’ se destrói na estrutura)

  x:~ (O objeto ‘x’ se destrói na estrutura)

- Exemplo de uma Ação Desligamento

  a:~ b (O objeto ‘a’ se desliga do objeto ‘b’)

  x:~ y (O objeto ‘x’ se desliga do objeto ‘y’)

B.2 Diagrama de Classes para Exceções

Nesta seção apresentamos o diagrama de classes contendo as exceções tratadas no SSO.

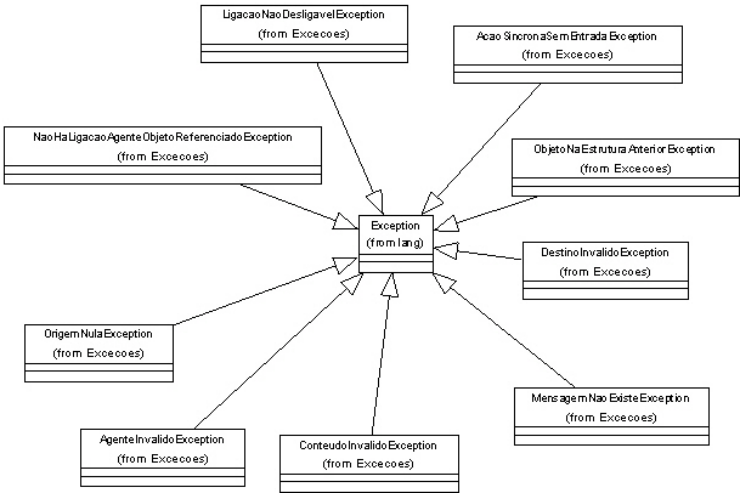


Figura B.1: Diagrama de classes contendo as exceções tratadas

B.3 Diagramas de Seqüência

O conjunto completo de diagramas definidos para o SSO é apresentado abaixo. Os diagramas estão relacionados à instanciação de elementos do SSO. Os diagramas foram construídos com base na notação UML para facilitar a interpretação.

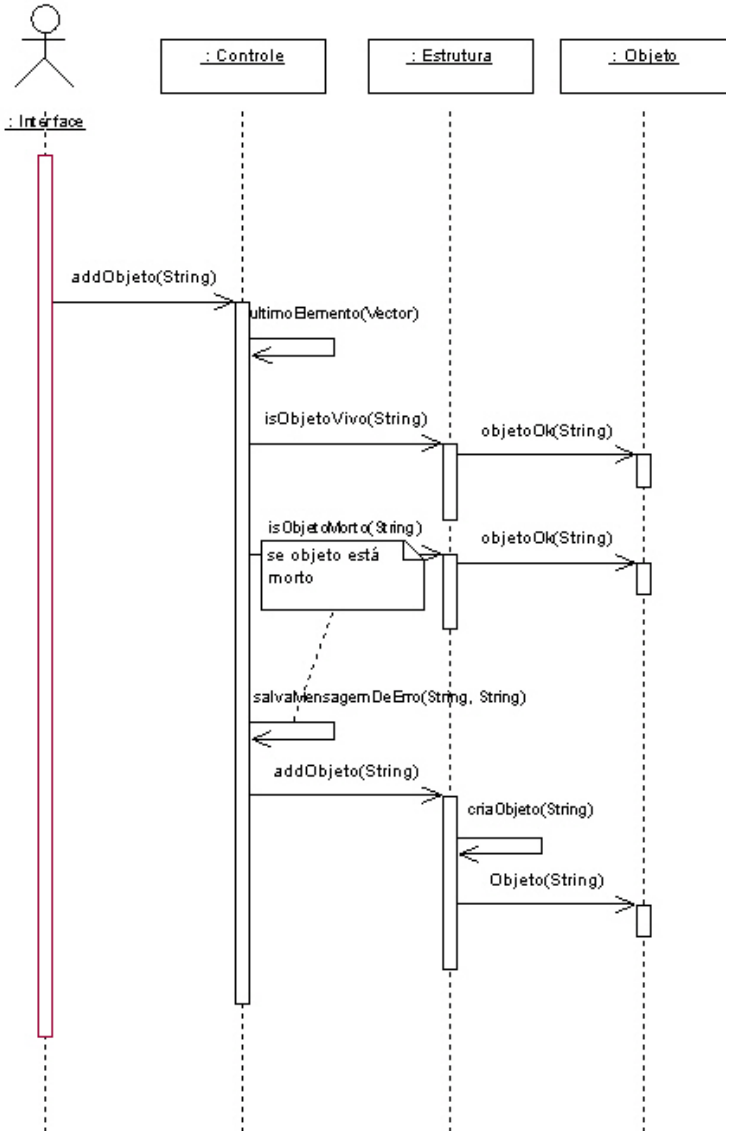


Figura B.2: Diagrama de seqüência *Definir Objeto*

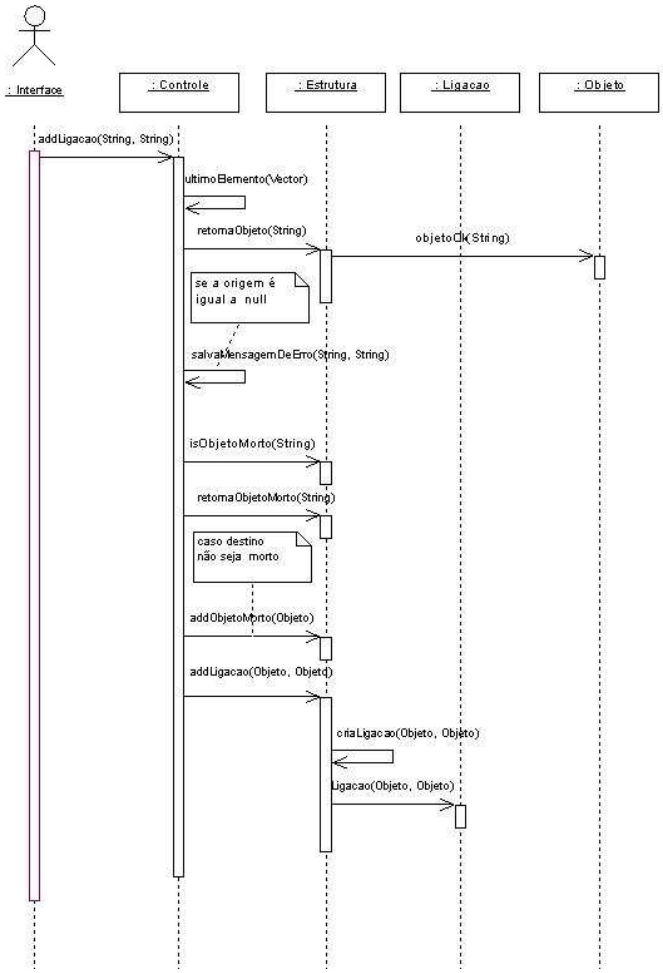


Figura B.3: Diagrama de seqüência *Definir Ligação*

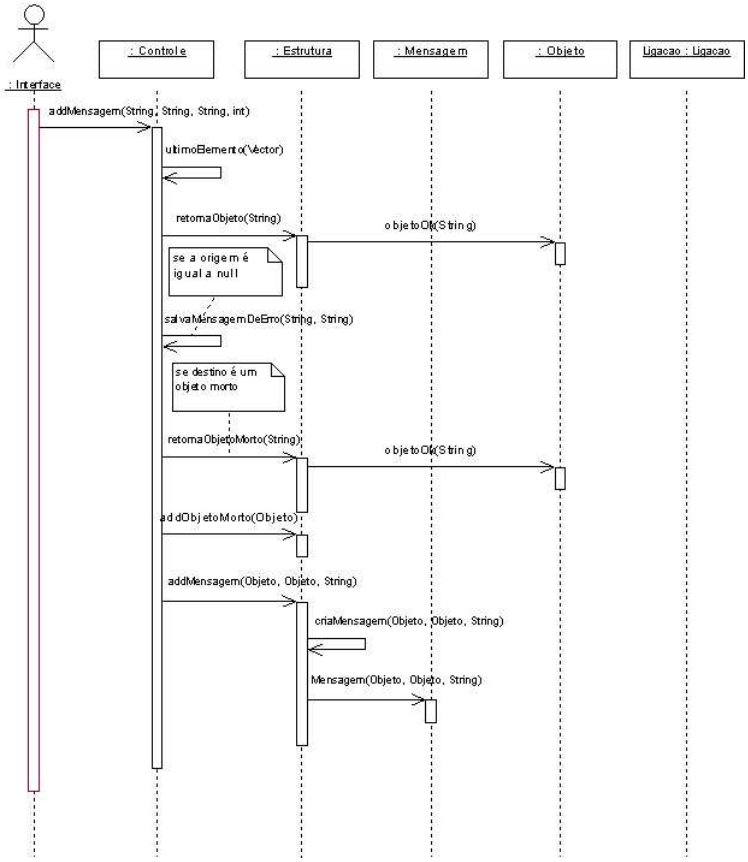


Figura B.4: Diagrama de seqüência *Definir Mensagem*

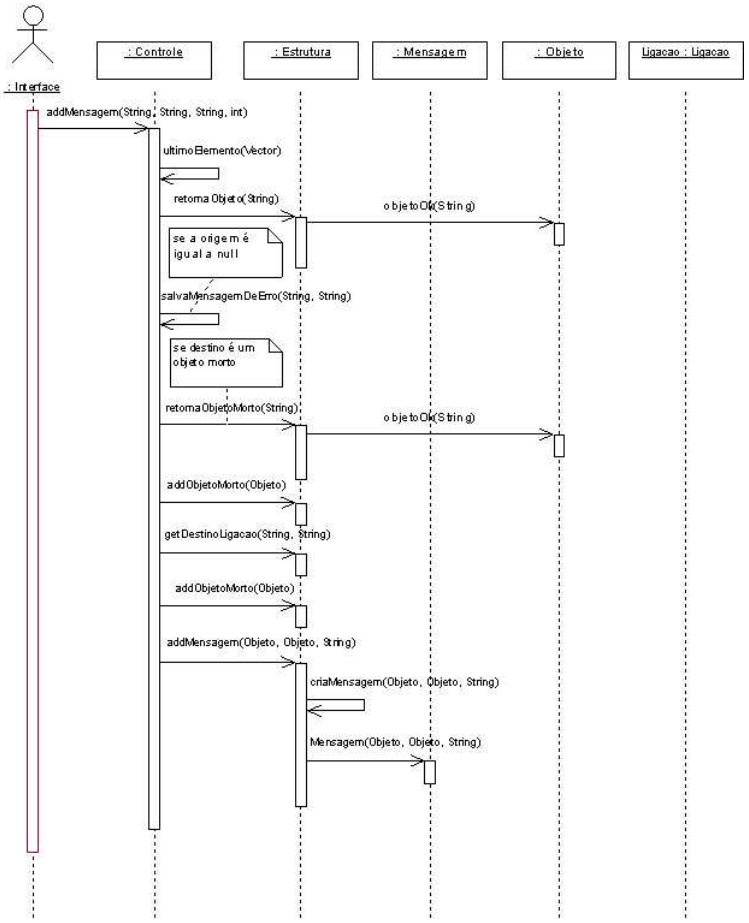


Figura B.5: Diagrama de seqüenciá *Definir Mensagem* (cujo conteúdo é uma referência)

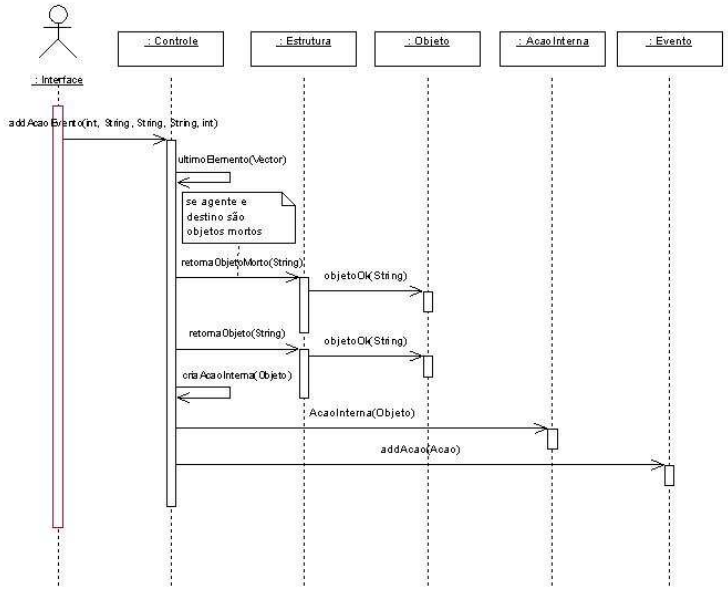


Figura B.6: Diagrama de seqüência *Definir Ação Interna*

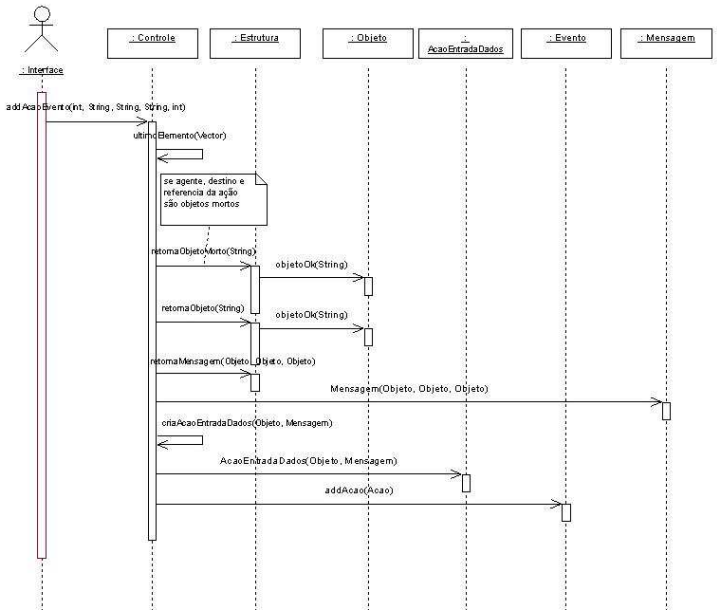


Figura B.7: Diagrama de seqüência *Definir Ação de Entrada de Dados* (conteúdo é uma referência)

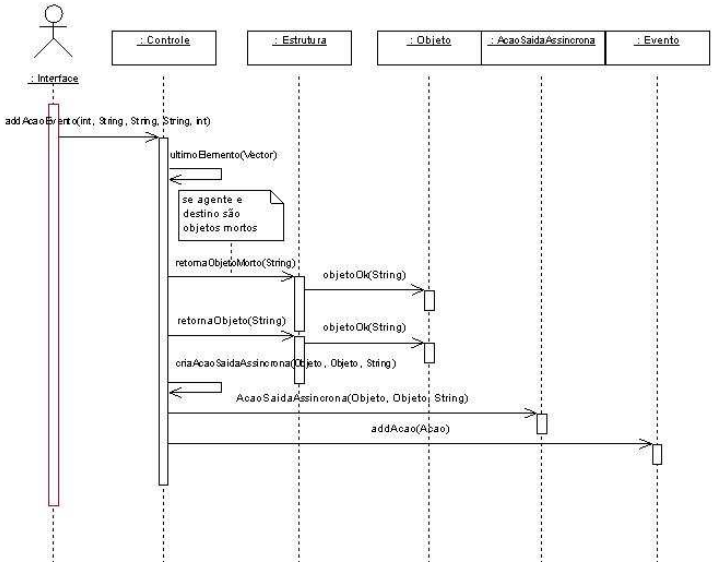


Figura B.8: Diagrama de seqüência *Definir Ação de Saída Assíncrona*

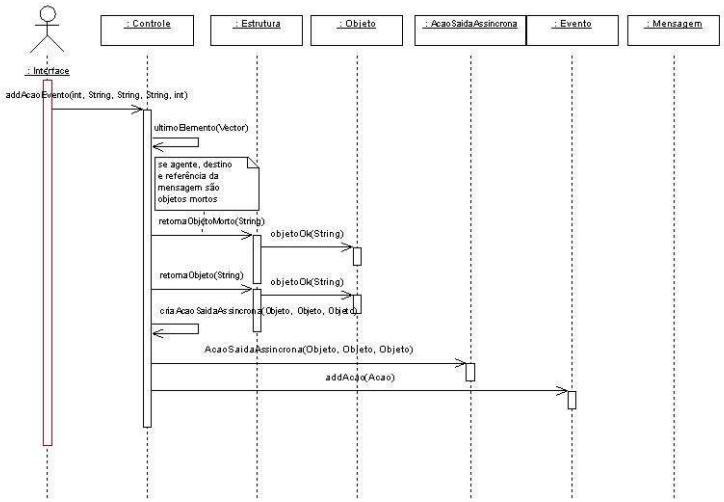


Figura B.9: Diagrama de seqüência *Definir Ação de Saída Assíncrona* (conteúdo da mensagem é uma referência)

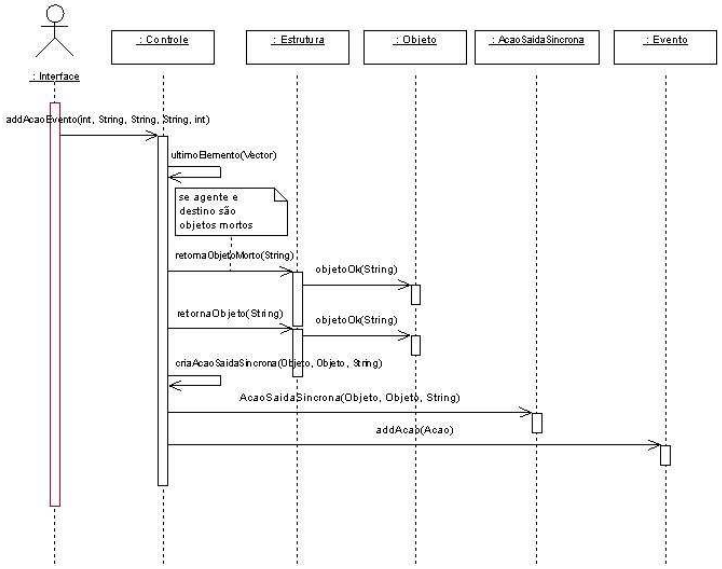


Figura B.10: Diagrama de seqüência *Definir Ação de Saída Síncrona*

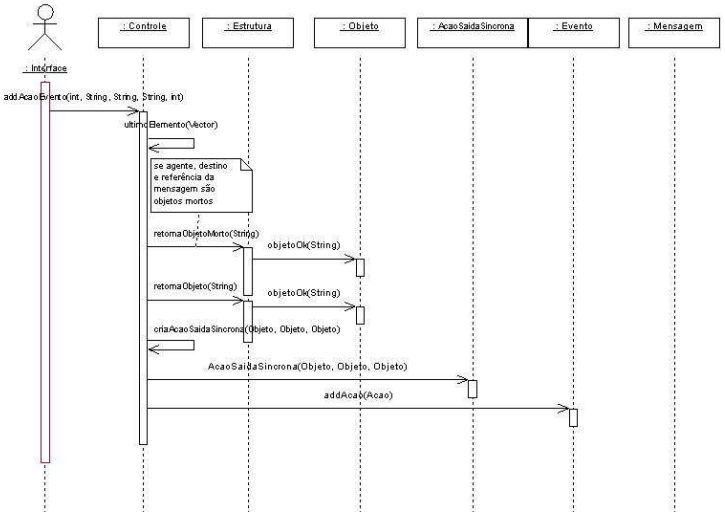


Figura B.11: Diagrama de seqüência *Definir Ação de Saída Síncrona* (conteúdo da mensagem é uma referência)

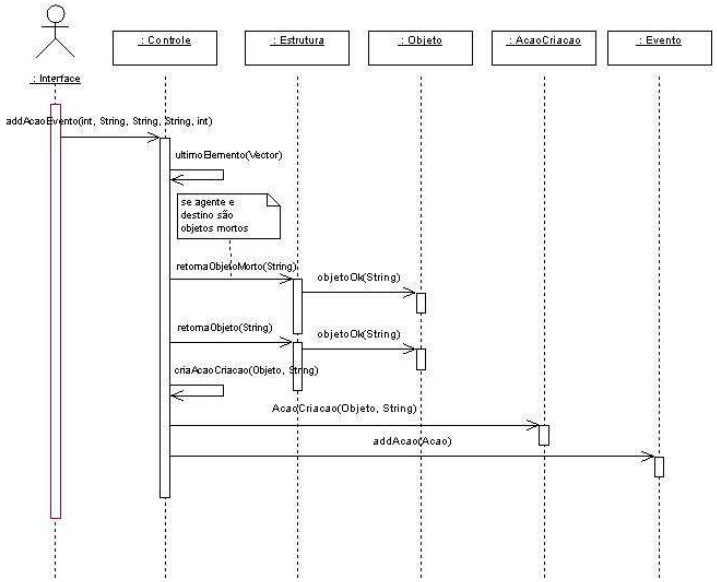


Figura B.12: Diagrama de seqüência *Definir Ação de Criação*

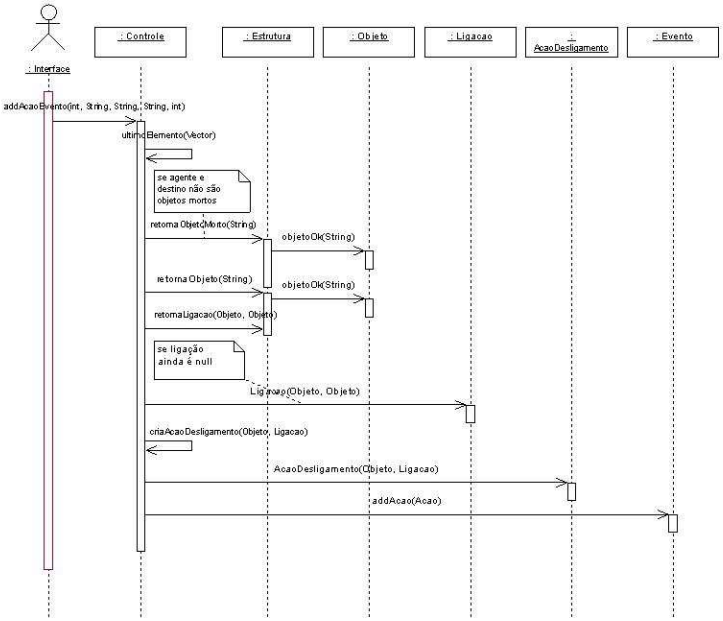


Figura B.13: Diagrama de seqüência *Definir Ação de Desligamento*

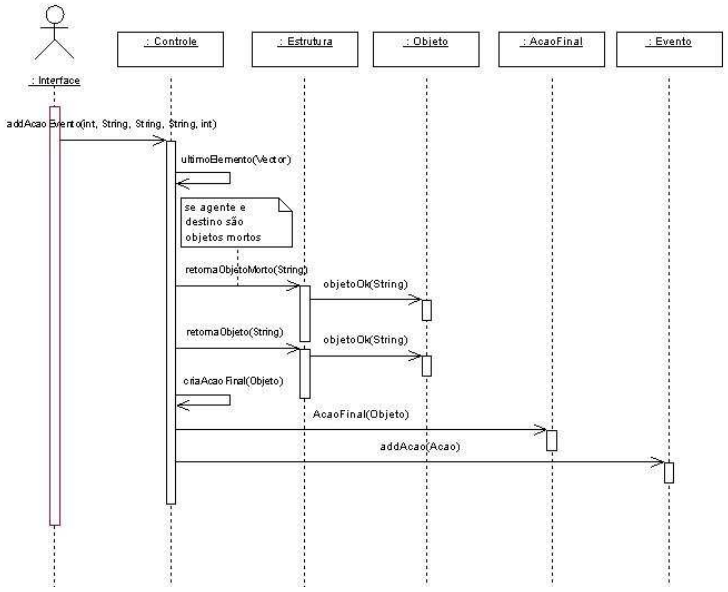


Figura B.14: Diagrama de seqüência *Definir Ação Final*

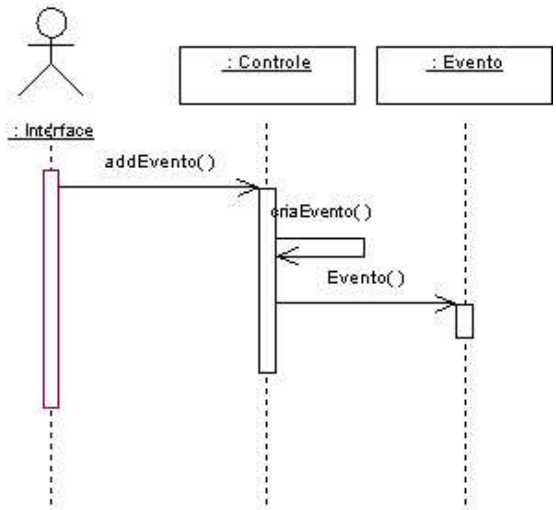


Figura B.15: Diagrama de seqüência *Definir Evento*