UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LOP: Uma Abordagem Unificada de Especificação Algébrica, Orientação a Objetos e Processos

por

Ausberto S. Castro Vera

Tese submetida como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Daltro J. Nunes
Orientador

Porto Alegre, 17 de novembro de 1995.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Castro Vera, Ausberto S.

LOP: Uma Abordagem Unificada de Especificação Algébrica, Orientação a Objetos e Processos / Ausberto S. Castro Vera. – Porto Alegre: CPGCC da UFRGS, 1995.

175 p.: il.

Tese (Doutorado)—Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1995. Nunes, Daltro J., Orient.

1.LOP. 2.Especificação Algébrica. 3.Objetos. 4.Processos. 5.Linguagem de Especificação. I. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Hélgio Casses Trindade
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Claudio Scherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. José Palazzo Moreira de Oliveira
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

AGRADECIMENTOS

A Deus pelo dom da vida, pela força e sustento de cada dia.

A minha esposa Elba, fiel companheira, pela compreensão, pelo ânimo nas horas difíceis, pela paciência, pelos sacrifícios e pelo carinho.

A meus filhos Weyden e Arthur, pela incansável espera das horas livres para 'ir ao parque ou ao centro'.

A Elizabeth, minha irmã, pelo apoio sacrificado nestes últimos cinco anos.

A meus pais e irmãos, pelas cartas e paciência, na esperança do meu pronto retorno a casa.

A meu orientador, Daltro, pelos conselhos, pelo apoio, pela força, pelo tempo dispensado e pela oportunidade de trabalhar juntos.

Ao amigo, Prof. Julio Claeyssen, pela amizade, pelo incentivo, pelos incontáveis e experimentados conselhos e pela ajuda na hora certa.

Ao CNPq, pela ajuda financeira, sem a qual teria sido impossível a realização dos estudos.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

A especificação abstrata de tipos de dados, é hoje um dos conceitos mais importantes, aceitos e compreendidos da Ciência da Computação, que permite descrever as principais entidades de um sistema baseado em computador através das propriedades que tais entidades devem satisfazer. Isto é feito usando métodos e linguagens algébricos, onde as propriedades são definidas na forma de axiomas (equações).

Por outro lado, a tecnologia chamada de Orientada a Objetos (OO), foi se transformando em uma disciplina amadurecida para projetos e implementações de aplicações de software. Atualmente esta tecnologia inclui muitas metodologias e muitas linguagens que abrangem todo o processo de desenvolvimento de sistemas, porém, a maioria delas são influenciadas pela implementação de tais sistemas, isto é, os conceitos básicos OO de classe, objeto e herança são defindos em função da linguagem de implementação a ser usada. Além disso, notamos que nos últimos anos está sendo desenvolvida muita pesquisa sobre uma geração de computadores que envolvem massivamente arquiteturas paralelas (computação concorrente), bem como sobre sistemas de comunicação de dados e engenharia (descrição) de protocolos.

O objetivo principal desta tese é dar uma resposta a estes três assuntos integrando três conceitos básicos da Engenharia de Software: Especificação Algébrica, Orientação a Objetos e Especificação de Processos e Concorrência, em uma única abordagem expressa através de uma Linguagem de Especificação Formal, chamada **LOP**. Esta linguagem é de natureza algébrica, com destaque para a semântica baseada em teorias em lógica de primeira ordem e na construção incremental de especificações baseada em bibliotecas.

PALAVRAS-CHAVE: LOP, Especificação Algébrica, Objetos, Processos, Linguagem de Especificação.

TITLE: "LOP: A UNIFIED APPROACH OF ALGEBRAIC SPECIFICATION, OBJECT-ORIENTATION AND PROCESSES"

ABSTRACT

The abstract specification of data types, one of the most important concepts accepted and understood of the Computer Science, allows to describe the the main entities of a based-computer system through the properties that these entities should be to satisfy. This is made using algebraic methods and languages, where the properties are defined as axioms (equations).

By other hand, the technology called Object-Oriented (OO), it has been transformed in a mature discipline for Design and Implementations of software applications. At present, this technology include many methodologies and many languages for the totality of the system development process. But the majority are influenced by the implementation of such systems, i.e., the basic concepts OO of class, object and inheritance are defined in accordance with the programming language to be used. Moreover, we noted that the last years are being developed many research on a computer generation that involve massively parallel architectures (concurrent computing) as well as on data communication systems and protocol engineering (description).

The main objective of this thesis is to give an answer to these three subjects integrating three basic concepts of Software Engineering: Algebraic Specification, Object Orientation and Processes and Concurrency specification, in an unique approach expressed through a language of formal specification, called LOP. This language has algebraic nature with prominence to the semantics based on theories in first-order logic with equality and the incremental construction of library-based specifications.

KEYWORDS: LOP, Algebraic Specification, Object-Orientation, Processes, Specification Language.

1 INTRODUÇÃO

1.1 Antecedentes

Hoje em dia não há mais dúvida da grande importância da especificação formal na Engenharia de Software, principalmente se consideramos que *um problema bem definido (bem especificado) está meio resolvido.*[1] Podemos mencionar algumas vantagens da especificação formal:

- Pode ser usada como documentação de um programa;
- Descreve as abstrações que estão sendo feitas;
- Pode ser usada para gerar verificação de propriedades e obtenção de novas propriedades.
- Pode ser considerada como um contrato entre os projetistas de um programa e seus clientes.
- É uma poderosa ferramenta para desenvolver uma parte de umprograma durante o processo de desenvolvimento do software.
- Em relação à validação de um programa, umaespecificação formal pode ser de muita ajuda para elaborarcasos de teste.

A especificação formal é caracterizada por ter um forte embasamento matemático. Os métodos formais podem ser orientados a *modelos* (objetos matemáticos estruturados pré-definidos) ou orientados a *propriedades* (na forma de axiomas). No atual estado da arte,o maior desenvolvimento tem sido no uso de certasestruturas matemáticas conhecidas como Álgebras. Daí o nomede Especificação Algébrica. Atual-

[1] A frase *A problem well defined is half solved* é um adágio antigo citado em [van 89a]

mente existem muitaslinguagens de especificação algébrica com avançadoconteúdo teórico e prático.

Na década de 80, uma nova abordagem foi desenvolvida para a construção de um sistema computacional nas suas diferentes etapas: análise, projeto e programação. Esta abordagem ou paradigma, conhecido como *Orientação a Objetos*, esta sendo incorporada aos poucos às especificações algébricas e não algébricas. Exemplos desta junção são as propostas Object-Z ([DUK 91]), Maude ([MES 90]), O=M ([IER 91], [IER 91a]) e outras. Object-Z é uma extensão da linguagem de especificação formal Z ([SPI 88]). Maude é uma linguagem baseada em uma lógica, chamada Lógica de Re-escrita e em OBJ3. A linguagem O=M é uma adaptação de VDM ([BJO 78], [BJO 88]) à abordagem orientada a objetos. Os elementos chaves do paradigma orientado a objetos são: objetos, classes, herança, encapsulamento e polimorfismo [TAK 90], [WEG 92].

Concorrência na especificação de sistemas ou especificação de sistemas concorrentes, é uma área também interessante cujo estudo teve seus inicíos nos trabalhos de R. Milner (CCS) e C.A.R. Hoare (CSP). Hoje existem muitos trabalhos avançados sobre concorrência, a maioria deles sob o ponto de vista teórico ou abstrato, por exemplo, álgebra de processos de J. A. Bergstra. Considerando que atualmente está sendo desenvolvida uma geração de computadores que envolvem massivamente arquiteturas paralelas (computação concorrente), outros estudos estão em andamento visando a implementação de concorrência pura, bem como, a combinação de concorrência e tipos de dados abstratos, concorrência e objetos. Trabalhos como RSL (Especificação algébrica + concorrência, [GEO 91]), Maude (lógica de re- escrita + objetos OBJ + concorrência, [MES 90]), SMoLCS-SCDS (Álgebras Dinâmicas, [AST 91], [AST 93], [AST 93a]), TDAs + concorrência ([KAP 87], [KAP 89]) e [HER 86] são alguns dos poucos exemplos do esforço para implementar concorrência algebricamente. Os elementos básicos da concorrência são os eventos e/ou ações, os processos e as operações sobre processos.

Uma característica comum a todas as linguagens mencionadas acima é que podem ser consideradas mais como linguagens de programação que como linguagens de especificação. As primeiras linguagens, usam os mesmos conceitos de uma linguagem de programação orientada a objetos. Por exemplo, um objeto é definido em função das propriedades dinâmicas que oferece um programa sendo executado (o comportamento de um objeto, a instância de uma classe, etc.).

Por outro lado, as linguagens que tratam com concorrência, caracterizam-se pelo uso dos conceitos de processo e evento a um nível baixo (pouca abstração) incorporando principalmente semântica operacional. Isto permite classifica-las também como linguagens de programação.

1.2 Objetivos da Tese

Considerando os antecedentes mencionados na seção anterior e interessados na pesquisa e desenvolvimento da área de Especificação Formal, nesta tese queremos desenvolver uma teoria que nos permita alcançar os seguintes objetivos:

☞-1 *A integração de três conceitos básicos: especificação algébrica, orientação a objetos e especificação de processos, em uma única abordagem expressada através de uma Linguagem de Especificação Formal*

Em relação à linguagem, significa que, em *primeiro lugar*, deve se fazer a escolha de uma linguagem algébrica já existente e utilizar toda ou uma parte da sua sintaxe e semântica, ou definir completamente uma nova linguagem (sintaxe e semântica novas). No caso de usar uma linguagem conhecida, sua sintaxe e/ou semântica pode ser uma extensão conservativa (que contém a anterior: as partes novas são criadas) ou uma modificação (algumas partes substituídas, eliminadas ou adaptadas).

Em *segundo lugar*, precisa-se adaptar os conceitos de classe e objeto a módulos (especificações) da linguagem algébrica. Uma característica que merece consideração do paradigma "orientado a objetos", é que este foi desenvolvido sem uma forte base em formalismo matemático. Para muitos profissionais de desenvolvimento de software (analistas, projetistas, programadores, etc.) o importante é que este paradigma oferece uma "solução alternativa" para a crise do software, sem a necessidade do uso de matemática sofisticada. Até agora não existe uma definiçãoestável e uniforme sobre o que é um objeto, sobre quaisconceitos são essenciais e quais opcionais. A maioria dasdefinições são informais. Vale a pena salientar também que,a maioria das definições que achamos na literatura "orientadaa objetos" tiveram sua origem nas linguagens de programaçãoorientadas a objetos, assim, uma definição de objeto ou classeestá relacionada a um determinado tipo de linguagem deprogramação. Em especificação formal, os conceitos edefinições não estão "amarradas" a como um objeto seráexecutado ou a como uma classe será uma implementada, e sim a uma abstração, por isso, poderão ser um pouco diferentes.

Por outro lado, as linguagens de especificação algébrica que já estão munidas de um certo grau de maturidade e uso, vem acompanhadas de uma forte fundamentação matemática. Neste contexto, falar de álgebras, sub-álgebras, sorts, signaturas, operações totais e parciais, ordem total e parcial, teorias, etc., é muito natural, porém ainda desconhecido em ambientes de desenvolvimento de software orientado a objetos. Outro aspecto a considerar aqui, é o mecanismo de herança: é necessário identificar ou unificar os conceitos de classes e subclasses do paradigma orientado a objetos, com os mecanismos de importação-exportação de módulos, módulos e submódulos, tipos e subtipos, sorts e subsorts, álgebras e sub-álgebras, relação de ordem, etc., das linguagens de especificação algébricas.

Em *terceiro lugar*, a linguagem proposta também será capaz de suportar concorrência. Em muitas teorias de concorrência é muito natural representar

processos como elementos de uma álgebra: uma expressão sobre o comportamento de sistemas concorrentes pode ser considerado como uma expressão sobre os elementos de alguma álgebra de processos. Existem diferentes estudos teóricos avançados sobre axiomatização de processos, porém o mais importante é o relacionado com as álgebras de processos realizado pelo grupo BBK (Projeto Esprit No. 432) do Center for Mathematics andComputer Science, Amsterdam. A linguagem a ser definida deveincorporar um conjunto mínimo de operadores sobre processos edefinir quais as limitações que determinarão o tipo deálgebra a ser usado. Por exemplo, já foram definidos algunstipos de álgebras como ACP (para processos comunicantes), AMP (como operador 'merge'), ASP (para processos síncronos), APC (com ooperador 'new' para a criação de processos), AFP_1 eAFP_2 (processos finitos), $prACP_I^-$ (processoscomunicantes probabilísticos), etc.. Uma tarefa que precisa ser realizada nesta parte, é estabelecer uma hierarquia de sistemas de axiomas que incorporam processos, de modo que seja possível definir o conceito de herança entre especificações que descrevam processos e seja fácil a sua reutilização para novas especificações.

Neste trabalho, foi escolhida a Linguagem Larch Compartilhada (LSL) do Projeto Larch [GUT 90], como ponto de partida para definir a sintaxe e semântica da parte algébrica da linguagem proposta. Entre as razões que forçaram esta escolha podemos mencionar as seguintes:

- A sintaxe de LSL é muito simples, fácil de entender e é considerada a parte fundamental para uso prático (ênfase na apresentação [GUT 85]).
- Não usa semântica baseada em álgebra inicial, facilitando a construção incremental de especificações incompletas.
- A construção de especificações é incremental eo ponto de partida é uma biblioteca de módulos LSL prontos.

- Regras como Indução e Redução fornecem uma poderosa ferramenta para raciocinar sobre a interpretação de uma especificação (operadores geradores, classes de equivalência, completeza suficiente, etc.).
- Existem muitas ferramentas associadas a LSL (editores de especificações, verificadores de sintaxe, provador de teoremas, etc.) que podem ser adaptados facilmente à linguagem proposta.

Algumas considerações *contra* a linguagem Larch Compartilhada que foram tomadas em conta para a definição de nossa proposta, mencionamos a seguir:

- LSL esta incluída dentro de uma família de linguagens num contexto bilíngüe especificação-programação: cada módulo LSL esta associado a outro módulo Larch/P e este por sua vez esta associado a um programa escrito em uma linguagem de programação P (forte dependência de uma linguagem de programação).
- Não existe uma preocupação séria pela semântica e sim pela sua sintaxe. Algumas declarações, como por exemplo **assumes**, não são muito claras para iniciantes em LSL. Isto torna a Linguagem Larch Compartilhada um pouco fraca.
- LSL foi desenhada para especificar unicamente sistemas seqüenciais.

☞-2 *Mostrar a Utilidade Prática da linguagem proposta através de exemplos e aplicações*

A maioria das linguagens de especificação formal, caracterizam-se pelo uso de elementos matemáticos para definir sua estrutura sintática e semântica. Isto facilita o uso de exemplos de natureza matemática: conjuntos, funções, tuplas, pilhas, etc., porém, isto não facilita que a linguagem seja usada para especificar problemas da vida real. Neste trabalho, apresentaremos na medida das possibilidades e com algumas limitações, exemplos práticos, que mostrem a utilidade real da linguagem.

☞-3 *Definir um Método associado à Linguagem definida*

As diferentes abordagens sobre a construção de um software, determinam um conjunto de etapas (em seqüência ou em paralelo) desde a apresentação do problema até a sua solução (o software sendo executado sem erros). Na prática, geralmente em cada uma destas etapas, são usadas metodologias diferentes e por especialistas diferentes. Visto que, uma destas etapas finaliza com a especificação do sistema a ser construído, nossa preocupação visa forçar a que esta especificação seja formal.

Considerando que as etapas anteriores à especificação são de natureza informal, tentaremos definir algumas tarefas necessárias para, a partir de um problema real, obter a especificação final na linguagem de especificação proposta. Nesta área vários métodos (operacional ou construtivo[2], algébrico ou abstrato[3], algorítmico[4], etc.) e linguagens (OBJ, LARCH, ASL, Axis, etc.) já foram desenvolvidos com resultados e aplicações muito importantes.

1.3 Estrutura do Texto

O texto da tese esta estruturado na seguinte maneira:

No Capítulo 2, é apresentada a Gramática e a Sintaxe da linguagem proposta (LOP). Podemos considerar este capítulo como o núcleo básico da tese, pois aqui são definidas as partes principais de uma especificação algébrica.

No capítulo 3, descrevemos a semântica da linguagem proposta, apresentando uma teoria em lógica multisortida de primeira ordem com igualdade como a semântica denotacional de uma especificação LOP.

[2] *Operacional* ou *construtivo* : especificações incluem aspectos de uma linguagem de programação imperativa, são mas detalhadas e a tendência é para sobre-especificação. Exemplos: Iota, RAISE.

[3] *Algébrico* ou *abstrato* : especificações usam somente fórmulas de lógica de predicados de primeira ordem, restritas a equações. Exemplos: OBJ, Larch, ASL.

[4] *Algorítmico* : Operacional + Algébrico. Exemplo: OBSCURE.

No Capítulo 4, são definidos os conceitos básicos que fazem parte da Orientação a Objetos adaptada a uma linguagem de especificação formal. Aqui ressaltamos a diferença entre Orientação a Objetos em uma linguagem de Programação e Orientação a Objetos em uma linguagem de Especificação. No Capítulo 5, é especificado os principais operadores das conhecidas Álgebras de Processos, como sistemas de axiomas, de modo que a linguagem proposta, nos permita especificar processos em um nível superior (mais abstrato).

No Capítulo 6, desenvolvemos um Método que permita, a partir de um problema real, construir especificações usando a linguagem proposta no Capítulo 2.

Na parte final do texto, é incluído dois anexos. O primeiro, apresenta muitas definições básicas que são necessárias para entender os diferentes conceitos dos capítulos anteriores. No segundo anexo, na forma de biblioteca, são apresentados um conjunto de especificações usadas nos diferentes exemplos do texto.

2 A LINGUAGEM LOP: SINTAXE

LOP é uma linguagem de especificação formal orientada a propriedades com as seguintes características principais:

- Incorpora três conceitos importantes da Engenharia de Software: especificação algébrica, orientação a objetos e especificação de processos e concorrência.
- Construção modular incremental de especificações: umas a partir de outras, tendo como base uma biblioteca compartilhada de especificações prontas.
- Todos os módulos são potencialmente parametrizados através do mecanismo de renomeação de sorts e operadores.
- A semântica de uma especificação é uma teoria em lógica multisortida de primeira ordem com igualdade.

Neste capítulo apresentaremos os aspectos relacionados com a sintaxe da linguagem. Conceitos sobre objetos, concorrência e semântica serão analizados nos próximos capítulos.

2.1 Sintaxe Livre-de-Contexto

Para melhor compreensão da gramática da linguagem LOP, adotamos a seguinte convenção sintática:

|	separador alternativo
{ *exp* }	*exp* é considerada uma unidade sintática
[*exp*]	a expressão *exp* é opcional
exp^*	zero ou mais *exp*
exp^*,	zero ou mais *exp* separadas por vírgulas
exp^+	uma ou mais *exp*
exp^+,	uma ou mais *exp* separadas por vírgulas

As palavras reservadas da linguagem aparecem em negrito, por exemplo, **class**, **union of**, **asserts**, etc.

specification	::=	class$^+$
class	::=	simpleId [({ name [: signature] }$^+$,)] : **class**
		{ shorthand | external }* opPart* propPart*
		[consequences]
name	::=	simpleId | opForm
opForm	::=	**if** _ **then** _ **else** _
		| [_] { simpleOp | logicalOp | eqOp } [_]
		| [_] openSym [placeList] closeSym [_]
		| [_] . simpleId
placeList	::=	_ { { sepSym | , } _ }*
signature	::=	sort*, $\longrightarrow$ sort
sort	::=	simpleId
shorthand	::=	enumeration | tuple | union
enumeration	::=	sort **enumeration of** simpleId$^+$,
tuple	::=	sort **tuple of** fields$^+$,
union	::=	sort **union of** fields$^+$,
fields	::=	simpleId$^+$, : sort
opPart	::=	**introduces** opDcl$^+$
opDcl	::=	name$^+$, : signature

propPart	::=	**asserts** genPartition* eqPart
genPartition	::=	sort { **generated** | **partitioned** } **by** operator+,
operator	::=	name [: signature]
eqPart	::=	{ **for all** varDcl+, eqSeq }*
varDcl	::=	simpleId+, : sort
eqSeq	::=	equation { eqSepSym equation }*
equation	::=	term [== term]
term	::=	logicalTerm | **if** term **then** term **else** term
logicalTerm	::=	equalityTerm { logicalOp equalityTerm }*
logicalOp	::=	$\wedge$ | $\vee$ | $\Rightarrow$
equalityTerm	::=	simpleOpTerm [eqOp simpleOpTerm]
eqOp	::=	= | $\neq$
simpleOpTerm	::=	simpleOp+ secondary | secondary simpleOp+
		| secondary { simpleOp secondary }*
secondary	::=	primary | [primary] bracketed [: sort] [primary]
bracketed	::=	openSym [term { { sepSym | , } term }*] closeSym
openSym	::=	[| { | $\langle$
closeSym	::=	] | } | $\rangle$
primary	::=	{ (term) | simpleId [(term+,)] }
		{ . simpleId | : sort }*
consequences	::=	**implies** { classRef*, genPartition* eqPart
		| [classRef+, genPartition+] eqSeq } conversion*
conversion	::=	**converts** operator+, [exempts]
exempts	::=	**exempting** [**for all** varDcl+,] term+,
simpleId	::=	alphaNumeric+
sepSym	::=	␣ (espaço em branco)
simpleOp	::=	alphaNumeric+
eqSepSym	::=	; | `<return>`

2.2 Estrutura Modular das Especificações LOP

Signaturas, declarações e/ou axiomas que logicamente compõem uma unidade (uma especificação) em LOP, são agrupados em um módulo (simples ou composto) chamado de *classe*. Uma classe pode descrever completamente um tipo de dado abstrato ou pode descrever um conjunto de propriedades compartilhadas por outros tipos de dados, via mecanismos de importação. Uma *especificação* LOP é construída a partir de um número finito de classes, e tem a seguinte forma sintática:

$$\text{LOP-}specification \;=\; class^{+}$$

onde

class = *class_name* : **class**

[{ **specialization of** *class_name* (... **for** ..., ...) } |

{ {**aggregation** | **composition**} **of** *class_name_list*}]

[*SortId* **enumeration of** $op_1, \ldots, op_n$]

[*SortId* **tuple of** $op_1 : SortId_i, \ldots op_n : SortId_k$]

[*SortId* **union of** $op_1 : SortId_i, \ldots op_n : SortId_k$]

[**introduces**

operations_part]

[**asserts**

[sort **{ generated** | **partitioned } by** *operators_list*]

[**for all** *variable_declarations*

axiom_part]

[**implies** [*axiom_list*]

[**converts** *operators_list*

[**exempting** *terms_list*]]]

Um módulo *simples* LOP tem a forma:

class_name : **class**

introduces

signatura (*operations_part*)

asserts

for all *variable_declarations*

axiom_part

e é denotado genericamente da seguinte maneira

$$\textbf{Spec} \overset{\text{notaç}}{=} \{\underbrace{S_1, \ldots, S_n}_{S}, \underbrace{op_1, \ldots, op_m}_{O}, \underbrace{ax_1, \ldots, ax_k}_{Ax}\}$$
$$\overset{\text{notaç}}{=} \{\underbrace{S, O}_{Sig}, Ax\}$$
$$\overset{\text{notaç}}{=} < Sig, Ax >$$

onde S é o conjunto de sorts que aparecem na especificação, e O é o conjunto de operadores que são restritos pelo conjunto de axiomas Ax. Sig nas linguagens algébricas é denotada também pelo símbolo Σ. Além disso, para um módulo LOP arbitrário $\textbf{Spec}_i$ denotaremos por Sig_i e Ax_i a signatura e o conjunto de axiomas respectivamente de $\textbf{Spec}_i$. Para entender o significado de alguns termos (sorts, signatura, operador, etc.) usados neste capítulo, recomenda-se a leitura da primeira parte do Anexo A-1.

Somente para fins de notação abreviada, doravante escreveremos

$$\textbf{Spec} = < Sig, Ax >$$

para denotar uma especificação qualquer.

Exemplo 2.1

Considere-se o módulo simples BUNCH(Bunch,Element), onde o sort Bunch (maço) representa uma generalização da idéia de conjuntos finitos, "bags", listas, etc.

BUNCH(Bunch,Element) : **class**

introduces

$$\begin{aligned} empty: & \ \{\} & \longrightarrow & \ \text{Bunch} \\ add: & \ \text{Bunch, Element} & \longrightarrow & \ \text{Bunch} \\ removeone: & \ \text{Bunch, Element} & \longrightarrow & \ \text{Bunch} \\ count: & \ \text{Bunch, Element} & \longrightarrow & \ \text{Nat} \end{aligned}$$

asserts

$$count(empty(), a) == zero() \qquad -BU1-$$

$$count(count(B, b), a) == if \ \ a = b \qquad -BU2-$$
$$then \ \ count(B, a) + 1$$
$$else \ \ count(B, a)$$

$$count(removeone(B, b), a) == if \ \ a = b \qquad -BU3-$$
$$then \ \ count(B, a) + 1$$
$$else \ \ count(B, a)$$

$$count(removeone(B, a), a) == if \ \ count(B, a) = 0 \qquad -BU4-$$
$$then \ \ 0$$
$$else \ \ count(B, a) - 1$$

onde *count* conta o número de ocorrências de um elemento dado em um maço, *removeone* remove uma ocorrência de um elemento a partir de um maço, *add* agrega um elemento a um maço, e *empty* define um maço vazio. A signatura Sig_{BUNCH} da classe BUNCH é formada pelo par <S,O> onde:

S = < Bunch, Element, Nat >

O = < empty, add, removeone, count >.

BUNCH pode ser escrito como

BUNCH = < Sig_{BUNCH}, {BU1, BU2, BU3, BU4}>

A cada especificação simples LOP está associado um *sort principal*, denotado por S_p, que caracteriza a especificação. O sort principal corresponde ao TOI (type of interest) definido por Guttag [GUT 78]. No exemplo acima, o sort principal é Bunch.

A parte que segue à cláusula **introduces** é a lista de operadores da especificação (cada um com sua signatura). Cada operador usado em uma classe deve ser declarado. As signaturas dos operadores,

$$op : s_1, \ldots, s_k \longrightarrow s_{k+1}$$

serão utilizadas na verificação de sorts nos termos onde os operadores são usados.

As restrições aos operadores são feitas por meio de equações. Uma equação em LOP é formada por dois termos do mesmo sort, separados pelo símbolo $==$

$$t_1 == t_2$$

e devem ser escritas após a cláusula **asserts**. Equações da forma

$$termo == true()$$

podem ser abreviadas pela inequação:

$$termo$$

Por exemplo, a especificação de uma *relação total* com uma única equação $(x \Re y)$ or $(y \Re x) == true()$ é escrito usando uma inequação (-TR-) na forma seguinte:

TOTALRELATION : **class**

introduces

$\_\Re\_$: T, T, $\longrightarrow$ Bool

asserts

for all x, y: T

$(x \Re y)$ or $(y \Re x)$ -TR-

2.2.1 Importação de Módulos

Importação de módulos em uma linguagem representa uma proteção adicional para projetistas e usuários de especificações confiáveis e de boa qualidade. A importação de módulos por uma classe LOP é feita usando as cláusulas

aggregation of *class_name_list*

e

composition of *class_name_list*

Quando uma classe S é a agregação ou composição de uma lista de classes $S_1, \ldots, S_n$, o conjunto de operadores de S é a união de todos os operadores de todas as classes $S_1, \ldots, S_n$ mais os operadores que aparecem na apresentação de S. Também, o conjunto de axiomas de S é a união dos conjuntos de axiomas de todas as classes importadas $S_1, \ldots, S_n$, junto com os axiomas presentes na apresentação de S.

Estas duas cláusulas estão relacionadas com Orientação a Objetos, e serão explicadas em forma detalhada no próximo capítulo.

2.2.2 Operadores Geradores

Os operadores definidos numa classe LOP, associados a um determinado sort S, são classificados em geradores (construtores primitivos e não primitivos), modificadores e observadores.

Os operadores *geradores* são aqueles que criam objetos de sort S

op_{ger} : Dom $\longrightarrow$ S.

Os operadores *modificadores* são aqueles que modificam (a estrutura de) objetos existentes

op_{mod} : Dom1, S, Dom2 $\longrightarrow$ S.

Os operadores *observadores* são aqueles que tomam objetos de sort S como entradas e retornam resultados de outro sort diferente de S

op_{obs} : Dom1, S, Dom2 $\longrightarrow$ T

onde T $\neq$ S.

A construção LOP

sortId **generated by** *gen_operators_list*

mostra um conjunto de operadores que são suficientes para gerar todos os valores

do sort *sortId*. A cláusula **generated by** tem embutido também o conceito de especificação suficientemente completa.

Uma especificação LOP é *suficientemente completa* com respeito à cláusula S **generated by** *Gen_Operators_List* se é possível mostrar que:
a). Todos os termos livres de variáveis de sort S que contém operadores de contradomínio S (op : Dom $\longrightarrow$ *S) são provadamente iguais a termos onde os únicos operadores com contradomínio S pertencem a Gen_Operators_List.*
b). Todos os termos livres de variáveis de sort diferente de S são provadamente iguais a termos que não *contém operadores com contradomínio S.*

A validade das duas condições acima, permitem ter um conjunto de axiomas suficientemente completo. Geralmente o sort S é sort principal da especificação.

Exemplo 2.2
A classe QUEUE(Q,E) é a especificação de uma fila de elementos genéricos

QUEUE(Q,E) : **class**

introduces

$newq$: { } $\longrightarrow$ Q
$isemptyq$: Q $\longrightarrow$ Bool
$appendq$: Q, E $\longrightarrow$ Q
$firstq$: Q $\longrightarrow$ E
$restq$: Q $\longrightarrow$ Q

asserts

Q **generated by** *newq, appendq*
for all q: Q, e: E

$isemptyq(newq()) == true()$ $\quad -Q1-$
$isemptyq(appendq(q,e)) == false()$ $\quad -Q2-$
$firstq(appendq(q,e)) ==$ **if** $isemptyq(q)$ $\quad -Q3-$
then e
else $firstq(q)$

$$restq(appendq(q, e)) == \textbf{if}\ \ isemptyq(q) \qquad -Q4-$$
$$\textbf{then}\ \ newq()$$
$$\textbf{else}\ \ appendq(restq(q), e)$$

implies

converts *firstq*, *restq*, *isemptyq*

exempting $firstq(newq()), restq(newq())$

Observa-se que os operadores geradores de Q são *newq* e *appendq*, isto é, qualquer termo de sort Q pode ser escrito usando somente os operadores *newq* e *appendq*. Assim, o termo

$$restq(appendq(restq(appendq(appendq(newq(), a1), a2)), a3))$$

de sort Q construído usando três operadaores: *restq*, *appendq* e *newq*, pode ser reduzido a outro termo (usando sucessivamente o axioma Q4), e ser escrito como

$$appendq(newq(), a3)$$

onde somente é necessário os operadores geradores *appendq* e *newq*.

2.2.3 Operadores Observadores

Operadores observadores (de um sort S) tomam, como argumentos, valores de S e retornam valores de outro sort (diferente de S). Estes operadores são usados para obter informações sobre termos de sort S, isto é, descrevem o comportamento funcional dos termos de S. Os operadores observadores são listados usando a cláusula

$SortId$ **partitioned by** *obs_operators_list*

A presença desta cláusula significa duas coisas:

a). Cada termo de sort $SortId$ é igual a algum termo onde *obs_operator_list* são os únicos operadores com domínio $SortId$.

b). *obs_operator_list* é um conjunto completo de observadores para o sort $SortId$: o conjunto *obs_operator_list* é suficiente para distinguir os termos diferentes de $SortId$. Se o conjunto

$$obs\_operator\_list = \{obs_1, \ldots, obs_n\}$$

e se as igualdades

$$obs_1(t_1) = obs_1(t_2)$$

$$obs_2(t_1) = obs_2(t_2)$$

.....

$$obs_n(t_1) = obs_n(t_2)$$

são todas válidas, então pode-se concluir que os termos t_1 e t_2 de sort $SortId$, são iguais.

Exemplo 2.3

Consideremos agora o exemplo anterior junto com a lista de operadores observadores

QUEUE(Q,E) : **class**

introduces

$$\begin{array}{rlll} newq & : \{\,\} & \longrightarrow & \mathrm{Q} \\ isemptyq & : \mathrm{Q} & \longrightarrow & \mathrm{Bool} \\ appendq & : \mathrm{Q},\ \mathrm{E} & \longrightarrow & \mathrm{Q} \\ firstq & : \mathrm{Q} & \longrightarrow & \mathrm{E} \\ restq & : \mathrm{Q} & \longrightarrow & \mathrm{Q} \end{array}$$

asserts

Q **partitioned by** *firstq, restq, isemptyq*

for all q: Q, e: E

$$isemptyq(newq()) == true() \qquad -Q1-$$

$$isemptyq(appendq(q,e)) == false() \qquad -Q2-$$

$$firstq(appendq(q,e)) == \textbf{if}\ isemptyq(q)\ \textbf{then}\ e\ \textbf{else}\ firstq(q) \qquad -Q3-$$

$$restq(appendq(q,e)) == \textbf{if}\ isemptyq(q)\ \textbf{then}\ newq()\ \textbf{else}\ appendq(restq(q),e) \qquad -Q4-$$

implies

converts $firstq, restq, isemptyq$

exempting $firstq(newq()), restq(newq())$

Os operadores observadores são três: $firstq$, $restq$ e $isemptyq$. O operador $isemptyq$ informa se uma fila é vazia, o operador $firstq$ informa quem é o primeiro elemento de uma fila, e o operador $restq$ devolve o resto de uma fila sem o primeiro elemento.

Como saber se dois termos q_1 e q_2 do sort Queue são iguais ou diferentes?. Por exemplo, suponhamos que

$$q_1 = appendq(appendq(appendq(newq(), a_1), a_2), a_3) =< a_1, a_2, a_3 >$$

$$q_2 = appendq(appendq(appendq(newq(), a_1), a_3), a_2) =< a_1, a_3, a_2 >$$

então, usando os operadores observadores, temos:

$$isemptyq(q_1) == false() == isemptyq(q_2) \qquad (*)$$

$$firstq(q_1) == a_1 == firstq(q_2) \qquad (**)$$

De outro lado,

$$r_1 = restq(q_1) == appendq(appendq(newq(), a_2), a_3)$$

$$r_2 = restq(q_2) == appendq(appendq(newq(), a_3), a_2)$$

$$first(r_1) == a_2 \neq a_3 == firstq(r_2)$$

implica que $r_1 \neq r_2$, logo q_1 é diferente de q_2. Observe-se que a igualdade parcial para q_1 e q_2 (* e **) não necessariamente implica a igualdade.

De manera análoga podemos usar os operadores observadores $isemptyq$, $firstq$ e $restq$ para provar que os termos

$$q_3 = appendq(restq(appendq(appendq(new(), a_1), a_2)), a_3)$$

e

$$q_4 = appendq(appendq(new(), a_2), a_3)$$

são iguais.

2.2.4 Conseqüências

Visto que, especificações LOP *não* são executáveis, os erros presentes em uma classe não poderão ser detectados através de testes automatizados. Isto não impede que algumas propriedades (axiomas) sejam verificadas usando outros mecanismos. Estes são implementados através de cláusulas escritas na parte final de cada especificação, e são usadas unicamente para propósitos de verificação sintática (não representa custo adicional à semântica da especificação).

• **Classes ou Axiomas Implícitos**

Assertivas sobre outras classes ou alguns termos do conjunto de axiomas de uma classe Spec e que podem ser verificadas, são feitas usando a cláusula

implies [*nameClass* [(... for ...)] *list*]

[*SortId* **generated by** *gen_op_list*]

[*SortId* **partitioned by** *obs_op_list*]

[*axiom_list*]

Exemplo 2.4

Na especificação QUEUE(Q,E) definida acima, a parte final :

implies

LINEARCONTAINER(*appendq* **for** *insert*)

indica que, uma conseqüência da definição da classe QUEUE(Q,E) é a definição implícita da classe LINEARCONTAINER (ver anexo A-2) com a única diferença que o operador *appendq* é substituído pelo operador *insert*, isto é, uma fila pode ser considerado como um "conjunto linear". Obviamente, esta conseqüência significa que a classe LINEARCONTAINER é previamente definida. Como veremos nos próximos capítulos, a re-definição da classe QUEUE pode ser substituída pela seguinte:

QUEUE(C,E) : **class**

specialization of LINEARCONTAINER(*appendq* **for** *insert*)

Exemplo 2.5

A classe SPARSE_ARRAY define um array de números inteiros:

SPARSE_ARRAY : **class**

aggregation of INTEGER, CARDINAL

introduces

new	:	{ }	$\longrightarrow$	Array
$assign$	:	Array, Int, Val	$\longrightarrow$	Array
$defined$	:	Int, Array	$\longrightarrow$	Bool
$\_[\_]$	:	Array, Int	$\longrightarrow$	Val
$isempty$	:	Array	$\longrightarrow$	Bool
$size$	:	Array	$\longrightarrow$	Card

asserts

Array **generated by** new, $assign$

Array **partitioned by** $defined$, $\_[\_]$

for all i, j : Int, v : Val, a : Array

$assign(a,i,v)[j] ==$ **if** $i = j$ $-SA1-$

then v

else $a[j]$

$\neg defined(i, new()) == true()$ $-SA2-$

$defined(i, assign(a,j,v)) == (\ i = j\ \vee\ defined(i,a)\ )$ $-SA3-$

$size(new()) == 0$ $-SA4-$

$size(assign(a,i,v)) ==$ **if** $defined(i,a)$ $-SA5-$

then $size(a)$

else $size(a) + 1$

$isempty(a) == (size(a) = 0)$ $-SP6-$

A afirmação de que um array com um elemento definido é não vazio, contido na classe acima, e que pode ser verificado, é feita incorporando a cláusula

implies for all a : Array, i : Int

$defined(i, a) \Rightarrow \neg isempty(a)$

Assim, muitos axiomas implícitos ou derivados de uma classe, são transformados em axiomas visíveis (explícitos) e verificáveis através da cláusula **implies ...** .

- **Completeza**

Outro tipo de conseqüências da definição de uma classe está associada à completeza da especificação. Usando a cláusula

converts *operators_list*

dizemos que os axiomas da classe definem totalmente o conjunto de operadores *operators_list*. Isto significa que não há necessidade de agregar à especificação nenhum outro axioma para definir completamente um operador contido em *operators_list*. O especificador poderá também querer definir um operador em forma parcial (incompleteza intencional) pela exclusão na especificação de alguns axiomas (ou termos). Para isto é usado a cláusula

exempting *term_list*

indicando que o conjunto de termos *term_list* deve ser 'excetuado' (excluído) na construção de axiomas para a especificação.

Exemplo 2.6

De novo, usamos a parte final da classe QUEUE(Q,E) já definida acima

converts *firstq, restq, isemptyq*

exempting *firstq(newq()), restq(newq())*

para indicar que os operadores *firstq, restq, isemptyq* são definidos totalmente (formam um conjunto definido completamente através dos axiomas presentes na especificação). Além disso, outra conseqüência dos axiomas Q1-Q4 é de que os operadores *firstq* e *restq* são definidos totalmente, porém, no domínio `Queue - {newq()}` (cláusula **exempting**).

2.2.5 Shorthands (compactos)

Shorthands são construções sintáticas que permitem apresentar especificações, em forma compacta e de fácil leitura, de conjuntos de operadores e axiomas comuns a muitas especificações. Podem ser definidas previamente de acordo às necessidades do especificador. Por exemplo, em [WIN 87], sã usados os compactos

B **record of** title: T, author : A, subject : S

U **record of** name: N, status: St, books: Bs

L **record of** users: Us, books: Lbs

para especificar livro, usuário e biblioteca respectivamente, como registros.

Os shorthands

$SortId$ **enumeration of** $op_1, \ldots, op_n$

$SortId$ **tuple of** $op_1 : SortId_i, \ldots op_n : SortId_k$

$SortId$ **union of** $op_1 : SortId_i, \ldots op_n : SortId_k$

foram definidos em [GUT 90] e por serem de uso frequente também são incorporadas implicitamente em LOP.

- O compacto *enumeration* define um conjunto de operadores constantes e um operador (sort) que enumera estas constantes.

  Cada cláusula da forma

  S **enumeration of** $c_1, \ldots, c_n$

  deve ser substituída por:

  introduces

  $c_1, \ldots, c_n : \{\} \longrightarrow$ S

  $succ$: S $\longrightarrow$ S

  asserts

  S **generated by** $c_1, \ldots, c_n$

  $(c_i \neq c_j) == true()$

  $succ(c_i) == c_{i+1}$

para $1 \leq i < j \leq n$.

Assim para "S **enumeration of** c_1, c_2, c_3, c_4" teremos o seguinte pedaço de especificação equivalente:

introduces

$c_1, c_2, c_3, , c_4 : \{\} \longrightarrow \mathrm{S}$

$succ : \mathrm{S} \longrightarrow \mathrm{S}$

asserts

S **generated by** $c_1, c_2, c_3, , c_4$

$(c_1 \neq c_2) == true()$

$(c_2 \neq c_3) == true()$

$(c_3 \neq c_4) == true()$

$succ(c_1) == c_2$

$succ(c_2) == c_3$

$succ(c_3) == c_4$

Exemplo 2.7

Queremos definir o sort `Funcionario` formado por cinco termos constantes. Para isto usamos a construção compacta

FUNCIONARIO : **class**

Funcionario **enumeration of** diretor, gerente, vendedor,

secretaria, officeboy

que é equivalente à classe (detalhada):

FUNCIONARIO : **class**

introduces

$$\begin{aligned}
diretor &: \{\} \longrightarrow \text{Funcionario} \\
gerente &: \{\} \longrightarrow \text{Funcionario} \\
vendedor &: \{\} \longrightarrow \text{Funcionario} \\
secretaria &: \{\} \longrightarrow \text{Funcionario} \\
officeboy &: \{\} \longrightarrow \text{Funcionario} \\
succ &: \text{Funcionario} \longrightarrow \text{Funcionario}
\end{aligned}$$

asserts

Funcionario **generated by** *diretor, gerente, vendedor, secretaria, officeboy*

$$diretor() \neq gerente() == true() \quad -FU1-$$
$$diretor() \neq vendedor() == true() \quad -FU2-$$
$$diretor() \neq secretaria() == true() \quad -FU3-$$
$$diretor() \neq officeboy() == true() \quad -FU4-$$
$$gerente() \neq vendedor() == true() \quad -FU5-$$
$$gerente() \neq secretaria() == true() \quad -FU6-$$
$$gerente() \neq officeboy() == true() \quad -FU7-$$
$$vendedor() \neq secretaria() == true() \quad -FU8-$$
$$vendedor() \neq officeboy() == true() \quad -FU9-$$
$$secretaria() \neq officeboy() == true() \quad -FU10-$$
$$succ(diretor()) == gerente() \quad -FU11-$$
$$succ(gerente()) == vendedor() \quad -FU12-$$
$$succ(vendedor()) == secretaria() \quad -FU13-$$
$$succ(secretaria()) == officeboy() \quad -FU14-$$

- O compacto *tuple* é usado para definir uma seqüência (tupla) de termos de comprimento fixo $< t_1, t_2, \ldots, t_n >$.

Cada cláusula da forma

S **tuple of** $f_1 : S_1, \ldots, f_n : S_n$

deve ser substituída por:

introduces

$[\_, \ldots, \_] : S_1, \ldots, S_n \longrightarrow$ S

$proj$: S, Nat $\longrightarrow S_k$

set_val : S, $S_k \longrightarrow$ S

asserts

S **generated by** $[\_, \ldots, \_]$

S **partitioned by** $proj$

for all s: S, $x_1, y_1 : S_1, \ldots, x_n, y_n : S_n$

$proj([x_1, \ldots, x_k, \ldots, x_n], k) == x_k$

$set\_val([x_1, \ldots, x_k, \ldots, x_n], y_k) == [x_1, \ldots, y_k, \ldots, x_n]$

para cada $1 \leq k \leq n$.

Exemplo 2.8

Deseja-se definir o sort (composto) `Endereco`. Isto é feito através da seguinte cláusula:

ENDERECO : **class**

Endereco **tuple of** lograd, bairro : String, cep : Nat, cidade : String,

uf: Str2

A classe equivalente é:

ENDERECO : **class**

introduces

$< \_, \_, \_, \_, \_ >$: String, String, Nat, String, Str2 $\longrightarrow$ Endereco

$\_\_.lograd$: Endereco $\longrightarrow$ String

$\_\_.bairro$: Endereco $\longrightarrow$ String

$\_\_.cep$: Endereco $\longrightarrow$ Nat

$\_\_.cidade$: Endereco $\longrightarrow$ String

$\_\_.uf$: Endereco $\longrightarrow$ Str2

set_lo : Endereco, String $\longrightarrow$ Endereco

set_ba : Endereco, String $\longrightarrow$ Endereco

set_ce : Endereco, Nat $\longrightarrow$ Endereco

set_ci : Endereco, String $\longrightarrow$ Endereco

set_uf : Endereco, Str2 $\longrightarrow$ Endereco

asserts

Endereco **generated by** $< \_, \_, \_, \_, \_ >$

Endereco **partitioned by** *.lograd, .bairro, .cep, .cidade, .uf*

for all l, l_0, b, b_0, d, d_0: String, c, c_0: Nat, u, u_0: Str2

$< l, b, c, d, u > .lograd == l \quad -EN1-$

$< l, b, c, d, u > .bairro == b \quad -EN2-$

$< l, b, c, d, u > .cep == c \quad -EN3-$

$< l, b, c, d, u > .cidade == d \quad -EN4-$

$< l, b, c, d, u > .uf == u \quad -EN5-$

$set\_lo(< l, b, c, d, u >, l_0) == < l_0, b, c, d, u > \quad -EN6-$

$set\_ba(< l, b, c, d, u >, b_0) == < l, b_0, c, d, u > \quad -EN7-$

$set\_ce(< l, b, c, d, u >, c_0) == < l, b, c_0, d, u > \quad -EN8-$

$set\_ci(< l, b, c, d, u >, d_0) == < l, b, c, d_0, u > \quad -EN9-$

$set\_uf(< l, b, c, d, u >, u_0) == < l, b, c, d, u_0 > \quad -EN10-$

No "shorthand" *tuple*, cada componente de um sort-tupla é incorporado em dois operadores. Assim, para o componente *cidade* temos os

operadores:

$\_\_.cidade$: Endereco $\longrightarrow$ String

set_ci : Endereco, String $\longrightarrow$ Endereco .

- A construção compacta *union* define um sort que corresponde a união disjunta de dois ou mais tipos de dados.

Cada cláusula da forma

S **union of** $f_1 : S_1, \ldots, f_n : S_n$

deve ser substituída por:

Stag **enumeration of** $f_1 : S_1, \ldots, f_n : S_n$

introduces

$f_j : S_j \longrightarrow$ S

$proj$: S, Nat $\longrightarrow S_j$

tag : S $\longrightarrow$ Stag

asserts

S **generated by** $f_1 : S_1, \ldots, f_n : S_n$

S **partitioned by** $proj$, tag

for all $x_1 : S_1$, ..., $s_n : S_n$

$proj(f_j(x_j), j) == x_j$

$tag(f_j(x_j)) == f_j$

para cada $1 \leq j \leq n$.

Exemplo 2.9

Queremos definir o sort `Empresa` como a união disjunta de grupos de pessoas que pertencem a categorias diferentes. Para isto usamos a classe FUNCIONARIO, já definida acima, e a construção:

EMPRESA : **class**

Empresa **union of** *diretor*: Dire, *gerente*: Gere, *vendedor*: Vend,

secretaria: Secr, *officeboy*: Boys

A classe ampliada é mostrada a seguir.

EMPRESA : **class**

Funcionario **enumeration of** $diretor, gerente, vendedor, secretaria, officeboy$

introduces

$$
\begin{array}{rcll}
diretor & : & \text{Dire} & \longrightarrow \text{Empresa} \\
gerente & : & \text{Gere} & \longrightarrow \text{Empresa} \\
vendedor & : & \text{Vend} & \longrightarrow \text{Empresa} \\
secretaria & : & \text{Secr} & \longrightarrow \text{Empresa} \\
officeboy & : & \text{Boys} & \longrightarrow \text{Empresa} \\
\_\_.dir & : & \text{Empresa} & \longrightarrow \text{Dire} \\
\_\_.ger & : & \text{Empresa} & \longrightarrow \text{Gere} \\
\_\_.ven & : & \text{Empresa} & \longrightarrow \text{Vend} \\
\_\_.sec & : & \text{Empresa} & \longrightarrow \text{Secr} \\
\_\_.boy & : & \text{Empresa} & \longrightarrow \text{Boys} \\
tag & : & \text{Empresa} & \longrightarrow \text{Funcionario}
\end{array}
$$

asserts

Empresa **generated by** $diretor, gerente, vendedor, secretaria, officeboy$

Empresa **partitioned by** $.dir, .ger, .ven, .sec, .boy$

for all d: Dire, g: Gere, v: Vend, s: Secr, b: Boys

$$diretor(d).dir == d \qquad -EM1-$$

$$gerente(g).ger == g \qquad -EM2-$$

$$vendedor(v).ven == v \qquad -EM3-$$

$$secretaria(s).sec == s \qquad -EM4-$$

$$officeboy(b).boy == b \qquad -EM5-$$

$$tag(diretor(d)) == diretor() \qquad -EM6-$$

$$tag(gerente(g)) == gerente() \qquad -EM7-$$

$$tag(vendedor(v)) == vendedor() \qquad -EM8-$$

$$tag(secretaria(s)) == secretaria() \qquad -EM9-$$

$$tag(officeboy(b)) == officeboy() \qquad -EM10-$$

Observa-se que cada componente é incorporado em três diferentes operadores. Por exemplo, para o componente *gerente* tem-se

$$gerente : \{\ \} \longrightarrow \text{Funcionario}$$

$$gerente : \text{Gere} \longrightarrow \text{Empresa}$$

$$\_.ger : \text{Empresa} \longrightarrow \text{Gere}\ .$$

3 A LINGUAGEM LOP: SEMÂNTICA

3.1 Semântica baseada em Álgebras

Para entender em forma detalhada os conceitos teóricos presentes neste capítulo, novamente recomendamos a leitura do Anexo A-1.

A maioria das linguagens de especificação algébrica que hoje circulam pelo mundo da especificação formal, incorporam semântica baseada em estruturas matemáticas chamadas de *álgebras*. Isto significa que cada especificação ou tipo de dado abstrato, pode ser interpretada através de uma álgebra. Em outras palavras, o conjunto de termos sem variáveis livres definidos pelos axiomas da especificação ou derivados destes, são validados em uma álgebra. Exemplos destas linguagens são: OBJ-3 [GOG 88], ASL [van 89], PSF$_d$ [MAU 89], OBSCURE [LEH 88], etc..

Existem dois tipos de álgebras que são utilizados para este propósito: álgebras multi-sortidas (MSA) e álgebras sortidas-ordenadas (OSA). Estas últimas são atualmente objeto de estudo por parte do grupo liderado por J. Goguen e J. Meseguer e implementadas na linguagem OBJ-3. As álgebras MSA e OSA são construídas a partir da signatura de cada especificação.

A linguagem LOP por ser de natureza algébrica, situa-se dentro da abordagem *orientada a propriedades*. Nesta abordagem, o comportamento de componentes de um sistema de software, é definido através de um conjunto de propriedades na forma de axiomas $\forall X(t_1 == t_2)$ [1], onde t_1 e t_2 são termos do mesmo sort e X é um conjunto finito de variáveis.

Seja **Spec** uma especificação LOP

$$\textbf{Spec} = < \text{Sig, Ax} >$$

[1] denotado também como (X, t_1, t_2)

onde Sig = < S, O > é a signatura, S é o conjunto de sorts que aparecem na especificação, $S_p \in S$ o sort principal associado a **Spec** e, O é o conjunto de operadores usados para construir o conjunto de axiomas

$$Ax = \{ \forall X_i(t_{i1} == t_{i2})/i = 1, n \}.$$

Usando a signatura Sig e o conjunto de variáveis

$$X = \{ X_s \ / \ s \in S\},$$

construe-se o conjunto (a linguagem) de Sig(X)-termos $T_\Sigma(X)$ usados para definir os "carregadores" A_s de uma álgebra A associada à especificação **Spec**.

Deve-se observar que para construir uma álgebra somente é necessário uma signatura Sig e não o conjunto de axiomas [WEC 92].

Para cada especificação existem muitas álgebras associadas e as mais importantes são aquelas que são classificadas sob determinados tipos de homomorfismos, resaltando as conhecidas como *álgebra inicial* e de *álgebra final* que formam classes cujas estruturas matemáticas indicam o significado dos axiomas da especificação.

Existem, porém, algumas objeções de natureza prática ao uso deste tipo de semântica em especificação formal, ainda que sua natureza teórica seja fundamentada matematicamente. Podemos considerar as seguintes:

- **Estrutura Complexa**

  O conjunto de álgebras que satisfazem os axiomas de uma especificação **Spec** é um modelo. Um subconjunto deste modelo são todas as álgebras iniciais. De cada álgebra inicial para outra álgebra do modelo existe um único homomorfismo. Todas as álgebras iniciais do modelo são isomorfas [GOG 78]. Em cada álgebra existe um conjunto carregador para cada sort e em cada conjunto carregador existe uma relação de equivalência que determina uma partição em classes de equivalência [van 89a].

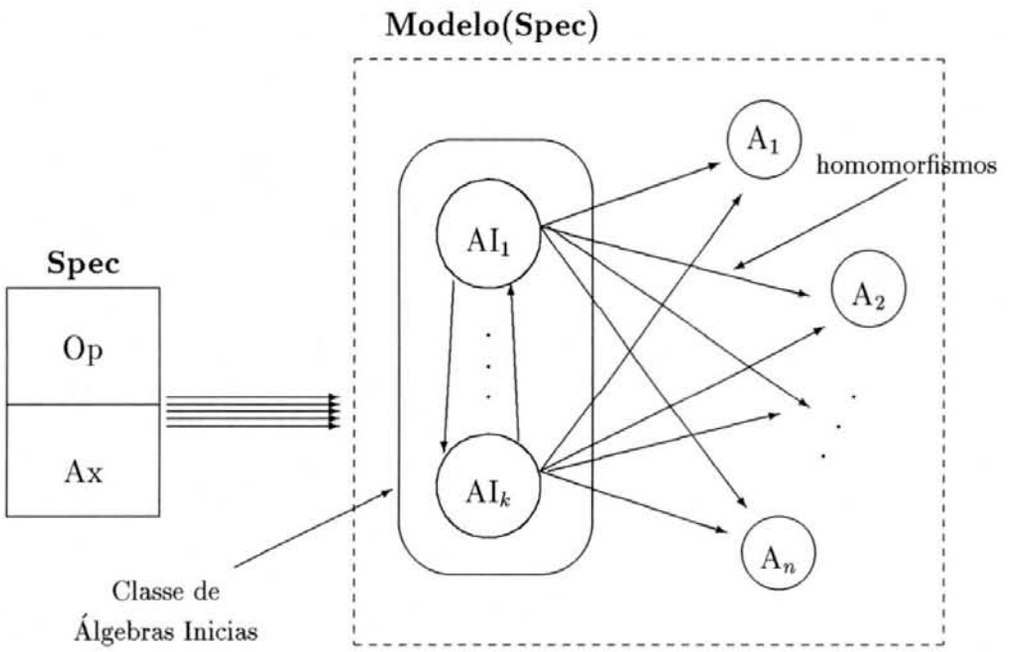


Figura 3.1 Semântica baseada em Álgebra Inicial

Toda esta estrutura matemática é muito difícil de ser "visualizada" na maioria dos casos práticos, principalmente em especificações complexas construídas incrementalmente. A pergunta, *como determinar o único homomorfismo entre duas álgebras para uma especificação dada?* não tem resposta fácil.

- **Modelo de uma Especificação**

  Existem duas correntes muito antigas sobre semântica de especificações baseadas em álgebras: *iniciais* (conjunto de termos de maior cardinalidade) e *finais* (conjunto de termos de menor cardinalidade). Pela quantidade de informação disponível sobre linguagens de especificação algébrica (ver [CAS 92]), podemos afirmar que a primeira tem tido melhor aceitação, porém, as duas interpretam uma especificação. Existe também o caso intermediário: muitas álgebras que não são nem iniciais nem finais podem dar a semântica de uma especificação (semântica "loose").

- **Especificações são Completas**

  Dois conceitos importantes em Especificação Formal são reusabilidade e construção incremental. Estes permitem construir especificações umas a partir de outras principalmente usando bibliotecas com módulos já verificados representando determinadas propriedades. Por outro lado, dependendo das necessidades do especificador, algumas especificações serão incompletas umas em relação a outras pela incorporação ou eliminação de algumas propriedades (axiomas). A abordagem algébrica (inicial ou final) somente é válida se a especificação é completa.

- **Problemas com o tratamento de Erros e Exceções**

  A semântica OSA foi desenvolvida pensando no problema de como construir especificações algébricas incorporando tratamento de erros e exceções. Problemas como o do construtor-seletor, representação múltiple, etc. estão sendo resolvidos nesta orientação, porém, com estruturas ainda mais complexas (ordem, subsort, etc.) [MES 93].

Como uma alternativa à abordagem algébrica, na seção seguinte apresentamos outro tipo de semântica baseada em teorias de lógica multisortida de primeira ordem.

3.2 Teorias em LOP

A especificação LOP de um objeto real qualquer descreve o conjunto de n propriedades que este objeto possui. Isto é feito através de um conjunto de sorts e operadores que satisfazem um conjunto finito de m axiomas.

Cada módulo (classe) LOP

$$\text{Spec} = < \text{Sig}, \text{Ax} >$$

onde Sig = < S, O >, define uma *teoria*, denotada por *Th (Spec)*, em lógica multisor-

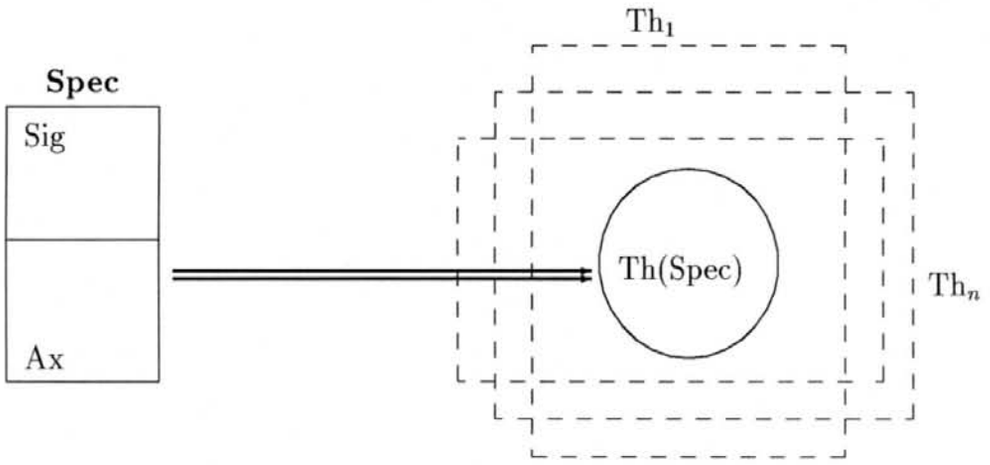


Figura 3.2 Teoria associada a uma especificação LOP

tida de primeira ordem com igualdade. Seja X um conjunto S-sortido de variáveis, então, uma *teoria associada* a um módulo LOP <Sig, Ax >, é o menor conjunto infinito (com respeito à inclusão de conjuntos) de Sig(X)-fórmulas sem variáveis livres (ver Figura 3.2), construída usando um alfabeto universal de símbolos para sorts, variáveis e operadores, que contém :

- as assertivas do módulo LOP (os axiomas),
- os axiomas convencionais da lógica de primeira ordem,
- o conjunto de fórmulas

  $$\{ \text{ if } true() \text{ then } x \text{ else } y == x,$$
  $$\text{if } false() \text{ then } x \text{ else } y == y \ \}$$

  para qualquer $x, y \in s$, para $s \in$ S.
- os axiomas derivados de Ax (usando as regras de inferência da lógica multisortida de primeira ordem).

- para cada cláusula "Sp **generated by** $op_1, ..., op_n$ ", para cada fórmula P e para cada variável $y \in Sp$, o fecho universal da fórmula de indução

$$(\forall y\, P) \equiv \bigwedge_{1 \leq i \leq n} \forall x_{i,1} \ldots \forall x_{i,k_i} \left(\left[\bigwedge_{j:sort(x_{i,j})=Sp} P[y \leftarrow x_{i,j}] \right] \Rightarrow P[y \leftarrow t_i] \right),$$

onde t_i é a expressão $\mathrm{op}_i(x_{i,1}, \ldots, x_{i,k_i})$ e as $x_{i,j}$ são variáveis distintas para um sort que não aparece em P. A expressão $sort(x_{i,j}) = Sp$ indica que o sort da variável $x_{i,j}$ deve ser Sp. (REGRA DE INDUÇÃO).

- para cada "Sp **partitioned by** $op_1, ..., op_n$ " e para cada par $y, z \in Sp$, o fecho universal da fórmula

$$(y = z) \equiv \bigwedge_{1 \leq i \leq n} \forall x_{i,1} \ldots \forall x_{i,k_i} \left(\bigwedge_{j:sort(x_{i,j})=Sp} t_i[x_{i,j} \leftarrow y] = t_i[x_{i,j} \leftarrow z] \right)$$

onde $t_i = op_i(x_{i,1}, \ldots, x_{i,k_i})$ e $x_{i,j}$ são variáveis diferentes de y e z e de sorts adequados. (REGRA DE REDUÇÃO).

- nada mais.

De acordo com isto, queremos dizer que, todas as fórmulas da teoria de um módulo LOP são derivadas somente da *presença* de assertivas no módulo e nunca da ausência de assertivas. Assim, uma teoria é a interpretação (semântica) de uma *apresentação* (signatura + axiomas) de uma especificação LOP, e não o contrário. Esta classe de semântica permite assegurar que

- a estrutura da semântica de uma especificação seja simples (um conjunto de fórmulas junto com uma relação de derivabilidade). Isto indica que o aspecto mais importante de uma especificação não é sua semântica e sim a sua sintaxe (apresentação) de modo que seja fácil de ler e usar.

- todos os teoremas provados sobre uma especificação incompleta (S_i), continuam válidos quando ésta é completada (S_c), isto é, Th(S_c) é uma *extensão conservativa*[2] de Th(S_i).

  Esta propriedade é conhecida como *monotonicidade*, visto que, as propriedades (os axiomas da especificação incompleta) são preservadas.

- as especificações são construídas formando hierarquias facilitando o mecanismo de herança.

Os componentes α, L, Λ e $\Re$ de uma teoria associada a uma especificação LOP

$$\text{Spec} = < Sig, Ax >$$

com Sig $= < S, O >$ são identificados da seguinte maneira:

- alfabeto α = S $\bigcup$ O
- linguagem L = conjunto de fórmulas construídas a partir de Sig. [3]
- axiomas Λ = o conjunto Ax de Spec.
- regras de inferência $\Re$ = regras de inferência da lógica multisortida de primeira ordem, mais as regras de indução e redução.

Exemplo 3.1

Suponha-se que queremos especificar um arquivo de Avisos de Pagamentos de uma determinada empresa. Este arquivo é um simples banco de dados onde são registrados os avisos de pagamentos remetidos pela companhia quando esta paga seus fornecedores.

[2]Um conjunto B é uma *extensão conservativa* de outro conjunto A, denotado por A$\ll$B ou A$\subseteq$B, se A é um subconjunto de B.

[3]Obviamente, Ax $\subseteq$ Th(Spec)

AVISO_PGTO : **class**

introduces

novo	:	{ }	$\longrightarrow$	Pagto
incluir	:	Pagto, AvPag, Data	$\longrightarrow$	Pagto
retirar	:	Pagto, AvPag	$\longrightarrow$	Pagto
$\_\_ \in \_\_$	:	AvPag, Pagto	$\longrightarrow$	Bool
vencim	:	Pagto, AvPag	$\longrightarrow$	Data

asserts

Pagto **generated by** *novo*, *incluir*

forall a, b: AvPag, d: Data, p: Pagto

$a \in novo() == false()$ $-PG1-$

$a \in incluir(p, b, d) == (a = b \ \vee \ a \in p)$ $-PG2-$

$retirar(novo(), a) == novo()$ $-PG3-$

$retirar(incluir(p, b, d), a) ==$ **if** $a = b$ $-PG4-$

then p

else $retirar(p, a)$

$vencim(incluir(p, a, d), b) ==$ **if** $a = b$ $-PG5-$

then d

else $vencim(p, b)$

implies converts *vencim*

exempting $vencim(novo(), a)$

Observamos que

S = { Pgto, AvPag, Data, Bool } $\quad S_p$ = Pgto

O = { *novo, incluir, retirar,* $\_\_ \in \_\_$ *, vencim* }

Sig = < S, O >

então podemos escrever AVISO_PGTO como

< Sig, {PG1, PG2, PG3, PG4, PG5} >

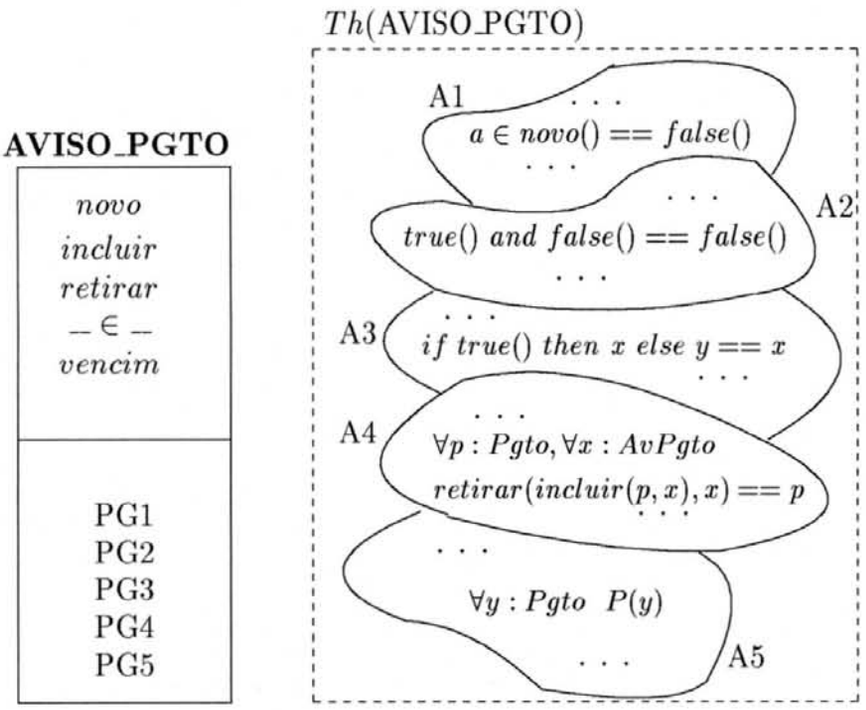


Figura 3.3 Teoria associada a AVISO_PGTO

A teoria associada a AVISO_PGTO (como mostrado na Figura 3.3) é determinada pela construção dos seguintes conjuntos de fórmulas:

1. O conjunto A1 de axiomas da especificação AVISO_PGTO

   A1 = {PG1, PG2, PG3, PG4, PG5}

2. O conjunto A2 de fórmulas $t_1 == t_2$ onde t_1 e t_2 são termos do conjunto {*true(), false(), b1 or b2, b1 and b2, not b1*} com $b1, b2 \in$ Bool.

   O conjunto A3 = { if true() then x else y == x,
   if false() then x else y == y }

   para x, y $\in s$, para $s \in$ { Pgto, AvPag, Data, Bool } .

3. O conjunto A4 de Sig(X)-fórmulas derivadas dos axiomas dos conjuntos A1, A2 e A3, como por exemplo

$$\forall p\text{: Pgto, } \forall x\text{: AvPag} \left(retirar(incluir(p,x),x) == p \right)$$

4. Para cada fórmula P e para cada $y \in$ Pagto, o fecho universal A5 da fórmula:

$$P[y \leftarrow novo()] \bigwedge \forall p\, \forall a \forall d \Big(P[y \leftarrow p] \Rightarrow P[y \leftarrow incluir(p,a,d)] \Big),$$

Assim,

$$\text{Th(AVISO\_PGTO)} = \text{A1} \bigcup \text{A2} \bigcup \text{A3} \bigcup \text{A4} \bigcup \text{A5}$$

3.3 Equivalência entre Álgebras e Teorias

Um aspecto importante de apresentar uma teoria como a semântica de uma especificação LOP, é que é possível provar que esta é equivalente à semântica baseada em álgebras, como sugere a fig. 3.4.

Consideremos uma especificação LOP

Spec $= < \Sigma, E >$, onde $\Sigma =< S, O >$. Seja X um conjunto S-sortido de variáveis e sejam A uma Σ-álgebra e $T_\Sigma(X)$ o conjunto de $\Sigma(X)$-termos construídos indutivamente a partir de Σ de **Spec**.

A equivalência entre álgebras e teorias é uma conseqüência de um conjunto de conceitos matemáticos derivados da Álgebra Universal aplicada a Ciência da Computação [WEC 92]:

1. Uma *atribuição* é um mapeamento

$$\alpha : \text{X} \longrightarrow \text{A}$$

2. Pelo *Princípio de Recursão Algébrico Finito* estabelece-se que uma atribuição α admite uma única extensão homomórfica

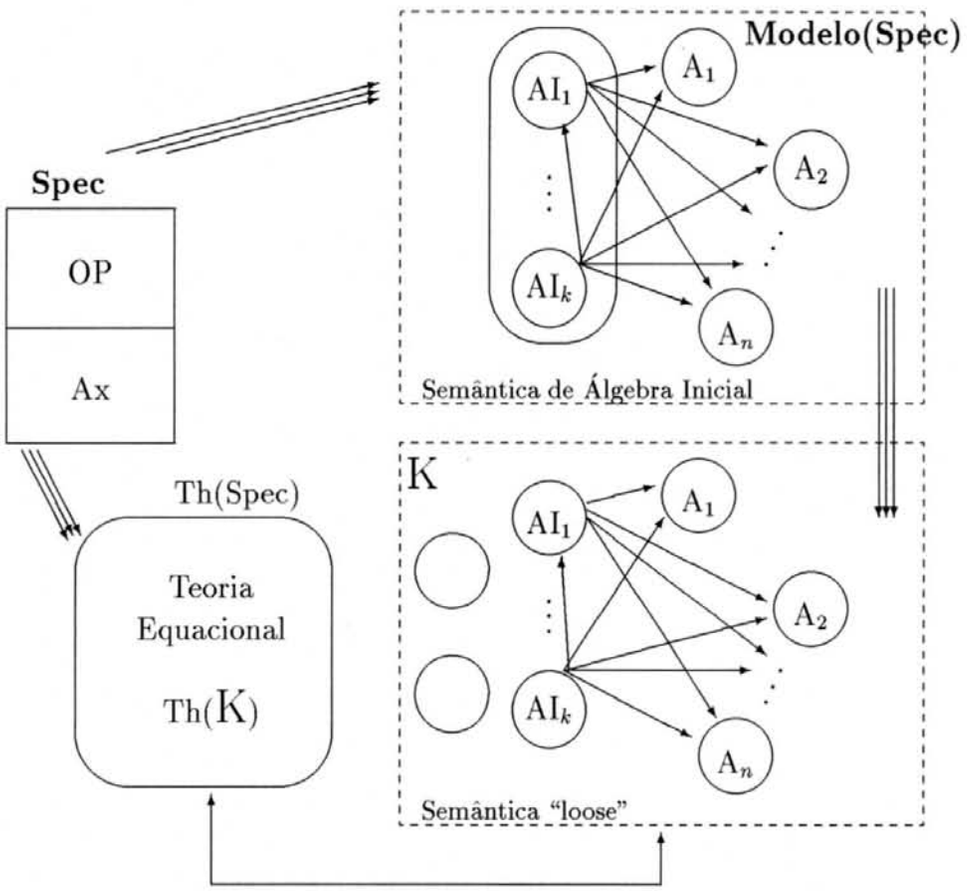


Figura 3.4 Semântica baseada em Álgebras e Teorias

$$\alpha^{\#} : T_{\Sigma}(X) \rightarrow A$$

onde $\alpha^{\#}(f(t_1, \ldots, t_n)) == f(\alpha^{\#}(t_1), \ldots, \alpha^{\#}(t_n))$ para $f \in \Sigma$.

3. Uma Σ(X)-equação (X, t_1, t_2), denotada por $\forall X(t_1 == t_2)$, é *válida* (ou satisfeita) em uma álgebra A, denotado por

$$A \models \forall X(t_1 == t_2)$$

se $\alpha^{\#}(t_1) == \alpha^{\#}(t_2)$ para toda atribuição α de X para A.

4. Uma Σ-álgebra A é um *modelo* do conjunto de $\Sigma(X)$-equações E, se cada Σ(X)-equação $\forall X(t_1 == t_2)$ de E é válida na álgebra A.

5. Uma classe K de álgebras é chamada *abstrata* se K é fechada sob álgebras isomórficas, isto é, A$\in$ K quando A é isomórfica a outra álgebra em K, i.é.

$$A \in K \Rightarrow \exists\, B \in K \text{ tal que } A \cong B$$

A classe de modelos de um conjunto de equações E (em particular E de **Spec**) é sempre abstrata.

6. Uma classe abstrata K de álgebras é chamada uma *classe equacional* se K é a classe de todos os modelos de algum conjunto de equações. Em particular, a classe abstrata K_{Spec} de todos os modelos do conjunto Ax de **Spec** é uma classe equacional.

7. Uma equação $\forall X(t_1 == t_2)$ é uma *conseqüência lógica* de um conjunto de equações E se $\forall X(t_1 == t_2)$ é válida em todos os modelos de E.

8. Seja K uma classe abstrata de Σ-álgebras. O conjunto

$$Th(K) = \{\forall X(t_1 == t_2) \;/\; A \models \forall X(t_1 == t_2), \quad A \in K\}$$

é chamado **Teoria** (equacional) de K.

Destes conceitos, decorre, em particular, o seguinte resultado:
se K é a classe equacional para Ax de **Spec**, *então*

$$\begin{aligned} Th(K) &= \{\forall X(t_1 == t_2) \in Ax \;/\; A \models \forall X(t_1 == t_2), \quad A \in K\} \\ &= Th(K_{Spec}) = Th(\mathbf{Spec}) \end{aligned}$$

A importância deste resultado é de caráter teórico, pois existe um conjunto de ferramentas matemáticas associadas a cada definição que sustentam a sua validade (ver o capítulo 3, em particular a seção 3.2 de [WEC 92]). Por outro lado, a sua importância é principalmente prática, desde que a construção de álgebras, isomorfismos e modelos, a partir de uma signatura e um conjunto de axiomas, é muito complicada e tediosa. Pelo contrário, como mostrado no exemplo acima, é muito mais prático e fácil construir uma teoria associada a uma especificação LOP.

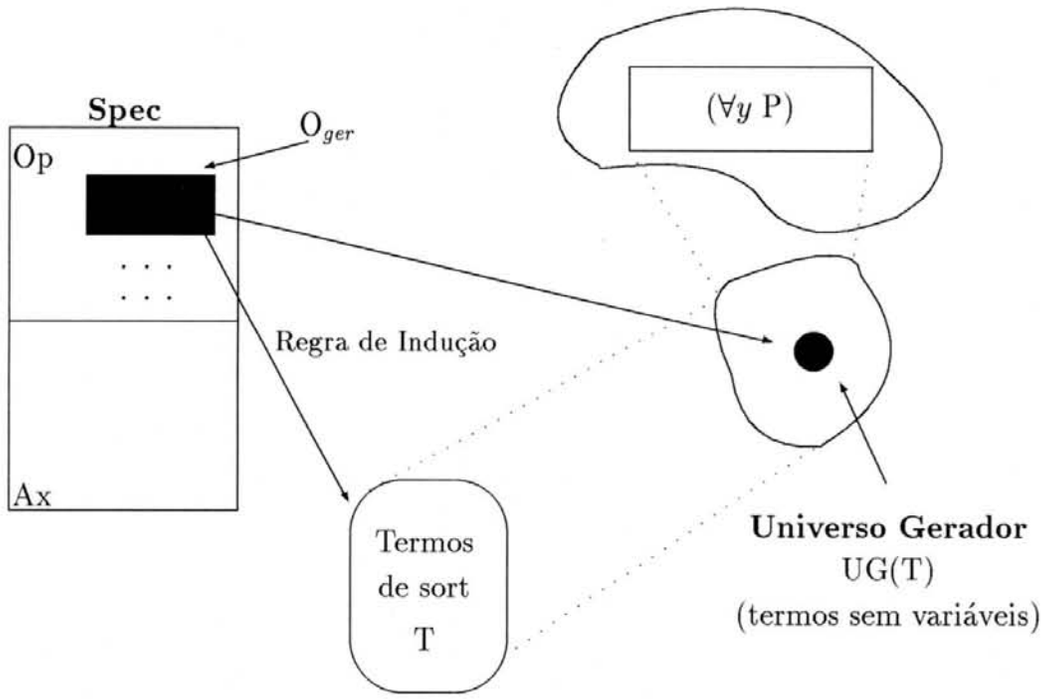


Figura 3.5 Regra de Indução em LOP

3.4 Indução, Redução e Conseqüências

3.4.1 Regra de Indução Geradora - Completeza Suficiente

Visto que, os operadores definidos em uma classe LOP podem ser classificados em geradores, modificadores e observadores, em relação a um sort (principal) T, a signatura desta classe é denotada também por

$$\text{Sig} = <\text{S}, \text{O}_{ger} \cup \text{O}_{mod} \cup \text{O}_{obs}>$$

onde $\text{O}_{ger} \cap \text{O}_{mod} = \emptyset$ e $\text{O}_{ger} \cap \text{O}_{obs} = \emptyset$. O conjunto O_{ger}, associado ao sort T e definido por

$$\text{O}_{ger} = \{ \text{g}_i : \text{Dom}_i \longrightarrow \text{T} \ / \ i = 1, \ldots, n \}$$

é chamado de *base geradora* para o sort T. Esta base geradora perimitirá *construir indutivamente* o conjunto de todos os termos de sort T.

O conjunto de termos de sort T é identificado com a teoria sobre o conjunto gerador O_{ger}. Podemos dizer que O_{ger} é uma *base geradora um-a-um* pelo fato de as expressões básicas sobre o conjunto O_{ger} estar em correspondência um-a-um com os termos de sort T (entenda-se melhor como, os valores do tipo T). O subconjunto de termos básicos (sem variáveis) sobre O_{ger} é chamado o *universo gerador* para o sort T e é denotado por UG(T).

O universo gerador para qualquer sort T é parcialmente ordenado pela relação *subtermo*, isto é, para $t_1, t_2 \in$ T a expressão $t_1 < t_2$ é válida se, e somente se, t_1 é subtermo de t_2.

Desde que, a relação subtermo é uma ordem noetheriana, "bem construída" em UG(T), então cada subconjunto de UG(T) tem um elemento mínimo. Esta informação permite aplicar sobre o Universo Gerador o *Princípio de Indução Estrutural* e logo aplicar o *Princípio de Indução Noetheriano* sobre o conjunto de termos construídos usando os operadores geradores. Ver Anexo A-1.

Pensando em aspectos de verificação e prova, este princípio indutivo é incorporado na linguagem LOP na forma de uma cláusula:

T **generated by** $op_1, \ldots, op_n$

onde $op_i \in O_{ger}$. Isto é chamado de *Regra de Indução Geradora* e é enunciada da seguinte maneira:

"Para cada cláusula T **generated by** *op_1, ..., op_n, e para cada fórmula P e cada variável y de sort T, a teoria associada a uma classe contém o fecho universal da fórmula de indução*

$$(\forall y\ P) \equiv \bigwedge_{1 \le i \le n} \forall x_{i,1} \ldots \forall x_{i,k_i} \left(\left[\bigwedge_{j:sort(x_{i,j})=T} P[y \leftarrow x_{i,j}] \right] \Rightarrow P[y \leftarrow t_i] \right)$$

onde t_i é a expressão $op_i(x_{i,1}, \ldots, x_{i,k_i})$ e as $x_{i,j}$ são variáveis distintas para um sort que não aparece em P "

Exemplo 3.2

Consideremos a definição de uma tabela que armazena valores em campos indexados

TABLE : **class**

aggregation of CARDINAL(Card)

introduces

new	:	$\{\ \}$	$\longrightarrow$	Table
add	:	Table, Ind, Val	$\longrightarrow$	Table
$\_ \in \_$	:	Ind, Table	$\longrightarrow$	Bool
$lookup$	:	Table, Ind	$\longrightarrow$	Val
$isempty$	:	Table	$\longrightarrow$	Bool
$size$	:	Table	$\longrightarrow$	Card

asserts

Table **generated by** new, add

Table **partitioned by** $\in$, $lookup$

forall i, j : Ind, v : Val, t : Table

$lookup(add(t,i,v),j) ==$ **if** $i = j$ $\quad -T1-$

then v

else $lookup(t,j)$

$i \in new() == false()$ $\quad -T2-$

$i \in add(t,j,v) == (i = j \ \vee \ i \in t)$ $\quad -T3-$

$size(new()) == 0$ $\quad -T4-$

$size(add(t,i,v)) ==$ **if** $i \in t$ $\quad -T5-$

then $size(t)$

else $size(t) + 1$

$isempty(t) == (size(t) = 0)$ $\quad -T6-$

Neste exemplo, os operadores geradores são op_1=new e op_2 = add. Suponhamos que queremos provar a validade da fórmula $(\forall t$: Table) P(t):

$(\forall t$: Table) $(isempty(t) \ \vee (\exists i$: Ind) $(i \in t))$

usando a regra de indução acima. Primeiramente observamos que esta fórmula não faz parte da apresentação de TABLE. Observamos também que temos dois tipos de

expressões $t_1 = new()$ (com zero variáveis-Table) e $t_2 = add(s, j, v)$ (com a variável-Table s que não aparece em P). Aplicando a fórmula de indução a P(t) temos:

$$(\forall t P(t)) \equiv \Big(\Big[P(t)[t \leftarrow] \Big] \Rightarrow P(t)[t \leftarrow new()] \Big) \ \bigwedge \ \forall s \Big(\Big[P(t)[t \leftarrow s] \Big] \Rightarrow P(t)[t \leftarrow add(s,j,v)] \Big)$$

Fazendo as substituições temos:

$$\equiv \Big(\Big[isempty(t) \ \vee \ \exists i : Ind \ (i \in t) \Big] \Rightarrow \underbrace{isempty(new())}_{axiomas\ T6+T4} \vee \exists i : Ind \ (\underbrace{i \in new()}_{axioma\ T2}) \Big) \ \bigwedge$$

$$\forall s \Big(\Big[isempty(s) \ \vee \ \exists i : Ind \ (i \in s) \Big] \Rightarrow \underbrace{isempty(add(s,j,v))}_{axiomas\ T5+T6+T2+T3} \vee \exists i : Ind \ (i \in add(s,j,v)) \Big)$$

$$\equiv \Big(\Big[isempty(t) \ \vee \ \exists i : Ind \ (i \in t) \Big] \Rightarrow true() \ \vee \ false() \Big) \bigwedge \forall s \Big(\Big[isempty(s) \ \vee \ \exists i : Ind \ (i \in s) \Big] \Rightarrow false() \ \vee \ true() \Big)$$

$$\equiv true() \ \bigwedge \ true()$$

isto é,

$$(\forall t P(t)) \equiv true()$$

Desta forma, usando a cláusula

Table **generated by** *new*, *add*

provamos (indutivamente) que a fórmula

$\forall t$: Table $(isempty(t) \ \vee \exists i$: Ind $(i \in t))$

é verdadeira.

Completeza Suficiente é uma noção de completeza descrito em [LIS 87] e associada à cláusula *generated by*. Esta define um conjunto de operadores que são suficientes para gerar todos os valores de um sort.

Uma especificação é completa suficientemente *com respeito à cláusula*

S **generated by** $op_1, \ldots, op_n$

se é possível mostrar que:

1. *Todos os termos livres de variáveis de sort S que contém operadores com contradomínio S são provadamente iguais a termos onde os únicos operadores com contradomínio S são* $op_1, \ldots, op_n$.
2. *Todos os termos livres de variáveis que não são de sort S são provadamente iguais a termos que não contém operadores com contradomínio S.*

Se estas duas condições valem, então a especificação têm axiomas suficientes.

Exemplo 3.3

Suponhamos que desejamos especificar um livro de aniversários, um simples banco de dados onde é registrado os aniversários das pessoas. Para isto definimos a seguinte classe:

LIV_ANIV : **class**

introduces

new	:	{ }	$\longrightarrow$	LivAniv
add	:	LivAniv, Name, Date	$\longrightarrow$	LivAniv
delete	:	LivAniv, Name	$\longrightarrow$	LivAniv
$\_ \in \_$	:	Name, LivAniv	$\longrightarrow$	Bool
lookup	:	LivAniv, Name	$\longrightarrow$	Date

asserts

LivAniv **generated by** *new*, *add*

forall n, m: Name, d: Date, t: LivAniv

$n \in new() == false()$ $\quad -BB1-$

$n \in add(t, m, d) == (n = m \ \vee \ n \in t)$ $\quad -BB2-$

$delete(new(), n) == new()$ $\quad -BB3-$

$delete(add(t, m, d), n) ==$ **if** $n = m$ $\quad -BB4-$

then t

else $delete(t, n)$

$lookup(add(t, n, d), m) ==$ **if** $n = m$ $\quad -BB5-$

then d

else $lookup(t, m)$

implies converts $lookup$

exempting $lookup(new(), n)$

Para mostrar que a especificação LIV_ANIV é completa suficientemente, precisamos mostrar que:

A.) Qualquer termo da forma $delete(t, n)$, onde t e n são termos livres de variáveis, é igual a um termo onde o operador *delete* não aparece.

B.) Qualquer termo da forma $n \in t$ ou $lookup(t, n)$, onde t e n são termos livres de variáveis, é igual a um termo onde *new*, *add* e *delete* não aparecem.

As duas condições acima podem ser provadas por indução sobre a profundidade de aninhamento, no termo t e provar que o operador *delete* pode ser eliminado. Primeiramente provaremos a parte A.). Em efeito:

- *Passo base:* Seja $t = new()$. Pelo axioma -BB3-, temos que

  $$delete(t, n) = delete(new(), n) = new()$$

  e assim, o operador *delete* pode ser eliminado.

- *Passo de Indução:* Suponha-se que, se a profundidade de aninhamento em t é menor que k, então o operador *delete* pode ser eliminado. Agora, seja o termo $add(t_1, n, d)$ com profundidade de aninhamento k. Pelo axioma -BB4-,

  $$delete(add(t_1, n, d), m) == t_1$$

  ou

  $$delete(add(t_1, n, d), m) == delete(t_1, m)$$

  A profundidade de aninhamento de t_1 e $delete(t_1, m)$ é menor que k de modo que, *delete* pode ser eliminado em ambos casos. Logo *delete*

pode ser eliminado do termo

$$delete(add(t_1, n, d), m).$$

e desta forma a prova tem sido completada.

Agora provaremos a parte B.) tanto para $n \in t$ quanto para $lookup(t, n)$. Isto significa provar que qualquer termo da forma $n \in t$, onde t e n são termos livres de variáveis, é igual a um termo onde *new*, *add* e *delete* não aparecem.

- *Passo base:* Seja $t = new()$. Pelo axioma -BB1-,

  $$n \in t = n \in new() == false()$$

  Em $false()$, os operadores *new*, *add* e *delete* não aparecem !.

- *Passo de Indução:* Suponha-se que, se a profundidade de aninhamento em t é menor que k, os operadores *new*, *add* e *delete* não aparecem. Agora, seja o termo $add(t_1, m, d)$ com profundidade de aninhamento k. Pelo axioma -BB2-,

  $$n \in add(t_1, m, d) == n = m == true() \ \vee \ false()$$

  ou

  $$n \in add(t_1, m, d) == n \in t_1 == true() \ \vee \ false()$$

  Em ambos casos, no termo $true() \ \vee \ false()$, os operadores *new*, *add* e *delete* também não aparecem.

E assim a fica provado para $n \in t$. Analogamente pode-se provar para o termo $lookup(t, n)$

3.4.2 Regra de Redução

Além da regra de indução, uma classe LOP pode incluir também uma regra que permite classificar todos os termos de um determinado sort. Isto é feito através de uma partição em classes de equivalência, de modo que seja possível de-

terminar quando dois termos são *distinguíveis*, isto é, quando dois termos representam valores diferentes. As *classes de equivalência* são determinadas pela relação de distinguibilidade e, cada uma representa um único valor. Para cada classe de equivalência existe um termo representativo (o gerador da classe). Dois termos t_1 e t_2 são equivalentes se pertencem a mesma classe de equivalência. Neste caso, diz-se que um pode ser reduzido ao outro e escrito $t_1 == t_2$. Para saber quando dois termos são distinguíveis, usa-se regras de re-escrita (redução), por exemplo, raciocínio equacional [van 89a]. Assim, se o sort Int representa os números Inteiros e, *zero*, *suc* e *pred* são operações associadas a Int,

$$zero : \{\ \} \longrightarrow \text{Int}$$
$$suc, pred : \text{Int} \longrightarrow \text{Int}$$

então:

$$0 = [zero()] = \{zero(), pred(suc(zero())), ...\}$$
$$1 = [suc(zero())] = \{suc(zero()), pred(suc(succ(zero()))), ...\}$$
$$2 = [suc(suc(zero()))] = \{succ(succ(zero())), ...\}$$

são classes de equivalência para os valores inteiros 0, 1 e 2 respectivamente. Os termos representativos para estas classes são $zero()$, $suc(zero())$ e $suc(suc(zero()))$ respectivamente. Também, o termo $pred(suc(zero()))$ pode ser reduzido ao termo $zero()$ por pertencer a mesma classe de equivalência.

A linguagem LOP usando a cláusula

S **partitioned by** $op_1, op_2, \ldots, op_n$

define uma partição de todos os termos do sort S (T_S) em classes de equivalência através de um conjunto completo de operadores observadores e construtores não primitivos $op_1, op_2, \ldots, op_n$. A *Regra de Redução* "S **partitioned by** $op_1, op_2, \ldots, op_n$" indica o seguinte: para qualquer par de termos t_1 e t_2 de sort S, se

$$op_1(t_1) == op_1(t_2) \quad \text{e}$$
$$op_2(t_1) == op_2(t_2) \quad \text{e}$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$op_n(t_1) == op_n(t_2)$$

então $t_1 == t_2$. Em outras palavras, a regra de redução indica que o conjunto infinito de termos de sort S pode ser particionado em um conjunto de classes de equivalência $\{[t_1], ..., [t_n], ...\}$, e onde qualquer termo t_j de uma classe $[t_i]$ pode ser reduzido ao termo t_i.

Exemplo 3.4

Desejamos saber quando duas pilhas de números inteiros são iguais. Para isto usamos a especificação de pilha:

STACK(C,E) : **class**

aggregation of INTEGER

introduces

$$\begin{array}{rcll} new & : & \{\ \} & \longrightarrow \text{ C} \\ push & : & \text{C, E} & \longrightarrow \text{ C} \\ pop & : & \text{C} & \longrightarrow \text{ C} \\ top & : & \text{C} & \longrightarrow \text{ E} \\ size & : & \text{C} & \longrightarrow \text{ Int} \\ isempty & : & \text{C} & \longrightarrow \text{ Bool} \end{array}$$

asserts

C **partitioned by** *top*, *size*, *pop*, *isempty*

for all s: C, e: E

$top(push(s,e)) == e$ $\quad - ST1 -$

$pop(push(s,e)) == e$ $\quad - ST2 -$

$size(new()) == 0$ $\quad - ST3 -$

$size(push(s,e)) == 1 + size(s)$ $\quad - ST4 -$

$isempty(new()) == true()$ $\quad - ST5 -$

$isempty(push(s,e)) == false()$ $\quad - ST6 -$

implies

converts *top*, *pop*, *size*, *isempty*

exempting $top(new())$, $pop(new())$

Sejam os termos:

$$s_1 = push(push(new(), 5), 8)$$

$$s_2 = push(push(new(), 7), 8)$$

$$s_3 = push(pop(push(push(new(), 5), 4)), 8)$$

dois termos-pilha que representam duas pilhas de números inteiros. Queremos saber se s_1 é igual a s_3, isto é, se s_1 pertence a mesma classe de equivalência de s_3. Usando a regra de redução temos:

$$top(s_1) == top(s_3) \qquad ('8')$$

$$size(s_1) == size(s_3) \qquad (2)$$

$$isempty(s_1) == isempty(s_3) \qquad (\ false()\ )$$

$$pop(s_1) == pop(s_3) \qquad (\ push(new(), 5)\ )$$

Como todas as igualdades são válidas, podemos concluir que os termos s_1 e s_3 podem ser reduzidos um ao outro, pertencendo os dois a mesma classe de equivalência. Agora, usaremos a regra de redução para o verificar se os termos s_1 e s_2 são equivalentes:

$$top(s_1) == top(s_2) \qquad ('8')$$

$$size(s_1) == size(s_2) \qquad (2)$$

$$isempty(s_1) == isempty(s_2) \qquad (\ false()\ )$$

Aparentemente, os dois termos-pilha s_1 e s_2 são iguais, porém, usando o operador construtor não-primitivo *pop*

$$pop(s_1) = push(new(), 5)$$

$$pop(s_2) = push(new(), 7)$$

observa-se que a igualdade $pop(s_1) == pop(s_2)$ não é válida. Isto significa que s_1 não pode ser reduzido a s_2 ou que os dois termos são distinguíveis (classes de equivalência diferentes).

A Regra de Redução

"S **partitioned by** $op_1, op_2, \ldots, op_n$ "

em uma classe LOP também pode ser expressada da seguinte maneira: para duas variáveis y e z de sort S, a teoria associada a esta classe contém o fecho universal

da fórmula

$$(y = z) \equiv \bigwedge_{1 \leq i \leq n} \forall x_{i,1} \ldots \forall x_{i,k_i} \left(\bigwedge_{j:sort(x_{i,j})=S} t_i[x_{i,j} \leftarrow y] = t_i[x_{i,j} \leftarrow z] \right)$$

onde $t_i = op_i(x_{i,1}, \ldots, x_{i,k_i})$ e $x_{i,j}$ são variáveis diferentes de y e z e de sorts adequados.

No exemplo acima, podemos demonstrar que $s_1 = s_3$, isto é, que o axioma $s_1 == s_3$ pode ser derivado dos axiomas da classe STACK(C,E), da seguinte forma:

$$\begin{aligned}(s_1 = s_3) \equiv \quad & \forall s \\ & \Big(top(s)[s \leftarrow s_1] = top(s)[s \leftarrow s_3] \quad \wedge \\ & \quad size(s)[s \leftarrow s_1] = size(s)[s \leftarrow s_3] \quad \wedge \\ & \quad pop(s)[s \leftarrow s_1] = pop(s)[s \leftarrow s_3] \quad \wedge \\ & \quad isempty(s)[s \leftarrow s_1] = isempty(s)[s \leftarrow s_3] \Big)\end{aligned}$$

Fazendo as substituições $[s \leftarrow s_1]$ e $[s \leftarrow s_3]$ na fórmula acima verifica-se que $s_1 == s_3$.

3.4.3 Conseqüências

Uma das propriedades da linguagem LOP, herdadas da linguagem LSL, está relacionada com algumas conseqüências, propriedades ou resultados implícitos presentes em muitas especificações. Afirmações da existência de um conjunto de propriedades implicítas dentro de uma classe LOP **Spec1**, é feito usando a cláusula **implies** nas seguintes formas:

- **implies Spec2**

  É usada para afirmar que dentro da classe Spec1 existe outra subclasse

Spec2, e que portanto, Th(Spec2) $\subseteq$ Th(Spec1). Por exemplo, ao especificar a classe INTEGER usamos a cláusula

implies NATURAL

para indicar que a todos os operadores e axiomas da classe NATURAL fazem parte da classe INTEGER.

- **implies T generated by list_operators_2**

  É usada esta cláusula para afirmar que um sort (principal) T em uma classe, pode ser gerado por duas listas diferentes de operadores: uma mencionada imediatamente após a cláusula **asserts** e, outra na cláusula **implies**

- **implies converts list_operators**

  Um problema comum em sistemas axiomáticos é decidir quando existem suficientes axiomas para definir totalmente um conjunto de operadores. A presença desta cláusula indica um certo tipo de completeza: os axiomas da classe definem totalmente qualquer operador desta lista, isto é, para qualquer operador da lista, se as interpretações de todos os operadores são fixados, existirá uma única interpretação para este operador. Em outras palavras, a cláusula indica que estes operadores são definidos totalmente, porém na base de outros operadores e que no momento da verificação devem ser convertidos. Por exemplo,

  $$pop(push(emptystack(), e)) == emptystack()$$

  .....

  implies converts *pop*

  indica que o operador *pop* é definido completamente, porém através do operador *emptystack* e quando da verificação da classe, deve-se converter todos os termos construídos com *pop*, neste caso, a $emptystack()$.

- **implies converts ... exempting ....**

  Existem casos em que algunos operadores, intencionalmente não são

definidos completamente, isto é, existem termos construídos com estes operadores, que não podem ser convertidos. Nestes casos, usa-se a cláusula **implies converts ... exempting .....** Este é um método de evitar situações de erro ou exceção, que deverão ser enfrentados somente quando se tratar de implementações.

Por exemplo, na especificação QUEUE(Q,E) mostrada anteriormente, a cláusula

implies converts $first$, $rest$, $isempty$

exempting $first(newq())$, $restq(newq())$

indica que o termo $first(newq())$ não pode ser convertido e que o operador $first$ é definido parcialmente.

Em qualquer dos casos acima, a teoria correspondente à uma conseqüência deve ser um subconjunto da teoria da classe onde a conseqüência aparece.

4 ORIENTAÇÃO A OBJETOS

No paradigma "Orientado a Objetos" (OO) existem três conceitos importantes:

- desenvolvimento orientado a objetos,
- linguagens orientadas a objetos e
- programação orientada a objetos.

Destes três conceitos, somente os dois primeiros são de interesse para a definição de LOP.

- *Desenvolvimento orientado a objetos:* a construção de um sistema orientado a objetos, a partir dos requerimentos, por uma análise, projeto e programação orientada a objetos.
- *Linguagem orientada a objetos:* uma notação bem definida que suporta propriedades orientadas a objetos (abstração, herança, ocultamento de informação, etc.) e especificação de um sistema orientado a objetos.

4.1 Classes e Objetos LOP

Os elementos básicos do paradigma "orientado a objetos" são os objetos, as classes e o mecanismo de herança.

O que que é um Objeto ? Uma Classe? Antes de dar uma definição final apresentamos algumas, encontradas na literatura:

- [COA 92]: Objeto é uma abstração de alguma coisa em um domínio de problemas, exprimindo as capacidades de um sistema de manter informações sobre ela, interagir com ela, ou ambos. Objeto é um encapsulamento de valores de Atributos e de seus Serviços exclusivos.

Classe é uma descrição de um ou mais Objetos por meio de um conjunto uniforme de Atributos e Serviços, incluindo uma descrição de como criar novos Objetos na Classe.

- [DUK 90]: Um **objeto** torna-se uma instância de uma classe. Cada **classe** consiste de (entre outras coisas) um estado junto com um número de operações relacionadas.

- [KOR 90]: **Objetos** são entidades básicas em execução de um sistema orientado a objetos. Objetos modelam as entidades no domínio da aplicação.

  Uma **classe** define um conjunto de possíveis objetos. Uma classe é um construtor para implementar tipos definidos pelo usuário. Idealmente, uma classe é uma implementação de um Tipo de Dado Abstrato.

- [PRE 87]: Um **objeto** é um componente do mundo real que é mapeado no domínio do software. No contexto de um sistema baseado no computador, um objeto é tipicamente um produtor ou consumidor de informação ou um item de informação.

- [RIN 92]: Um **objeto** é uma abstração de uma entidade do mundo real. **Classe** é um conjunto ou coleção de objetos tendo características comuns.

- [SAL 92]: Os **objetos** são abstrações de dados do mundo real, com uma interface de nomes de operações e um estado local que permanece oculto. O *tipo* de objeto pode ser visto como a descrição ou especificação de objetos possuindo a mesma estrutura e operações.

  Um conjunto de objetos que possui o mesmo tipo (atributos, relacionamentos, operações) pode ser agrupado para formar uma **classe**. A noção de classe é associada ao tempo de execução. Cada classe tem um tipo associado.

- [TAK 90]: Um **objeto** é um ente independente composto por: *estado interno* (uma memória interna em que valores podem ser armazenados e modificados ao longo da vida do objeto) e *comportamento* (um conjunto de ações pré-definidas (métodos) através das quais o objeto responderá à demanda de processamento por parte de outros objetos).

  Objetos de estrutura e comportamento idênticos são descritos como instâncias de **classes**

- [WEG 92]: **Objetos** são as unidades atômicas de encapsulamento. Objetos particionam o estado de uma computação em pedaços encapsulados. Cada objeto tem uma interface de operações que controlam o acesso a um estado encapsulado.

  Classes especificam o comportamento de coleções de objetos com operações comuns. Classes gerenciam coleções de objetos.

- [WIR 90]: Um **objeto** expressa uma abstração. Fornece serviços a seus clientes. Objetos podem compartilhar implementações. Uma implementação para muitos objetos (instâncias) é chamada uma **classe**.

- [WOL 90]: Um **objeto** é uma entidade computacional que pode encapsular comportamento e estado, e interatua enviando e recevendo mensagens.

As definições acima mrecem dois comentários. Em *primeiro lugar*, a maioria delas são *informais*: suas fontes são abordagens informais dentro do paradigma orientado a objetos. Entende-se um conceito ser *formal* se esta associado a algum conceito ou fundamento matemático. Se tentarmos formalizar algumas destas definições, podemos chegar a ambigüidades, por exemplo, segundo [RIN 92] e [WEG 92], objetos poderiam ser considerados como os *elementos* (atômicos) de um conjunto atômico (sem estrutura). Porém segundo [TAk 90] e [SAL 92], objetos seriam os elementos (pares ordenados <est, comport>) de um conjunto estruturado

(produto cartesiano Estado × Comportamento). No caso da definição de *classe*, a idéia da maioria dos autores corresponde a idéia matemática de um conjunto (elementos + propriedades comuns).

Em *segundo lugar*, muitas das definições acima, estão associadas a um determinado conceito de implementação (uma máquina + uma linguagem de programação) onde a maneira como os objetos são criados ou executados é mais importante. Daí a idéia de que os objetos são entidades dinâmicas, que podem ser criados ou destruídos dinamicamente durante o processo de execução de um programa.

Considerando que LOP é uma linguagem de especificação e que os elementos de estudo são principalmente de natureza abstrata, as definições de objeto e classe estão associados a este contexto, sendo o aspecto mais importante o que tem a ver com o formalismo usado e a abstração.

Um *objeto* (simples ou composto) é um componente do mundo real que pode ser representado por um elemento no domínio do software [PRE 87]. A parte do domínio do software que é de nosso interesse são as especificações LOP. Como o domínio do software pode ser matematicamente representado por subconjuntos definidos de acordo as características do seus elementos, então definimos formalmente um **objeto** como

um elemento *de um determinado* conjunto

e este será chamado de **classe**. Significa que, a formação de classes é feita no mundo real e logo estas devem ser mapeadas para o mundo abstrato. Por exemplo, as classes reais (concretas) onde os elementos são as capas de uma revista ou livro, portas e janelas de uma casa, quadro negro, tela do computador, frame de um editor, envelope de um gráfico, etc. classificadas em função do atributo "4 lados" são mapeadas na classe abstrata *window* (quatro pontos)

$$\{< x_1, y_1, x_2, y_2 > \ / \ x_1, y_1, x_2, y_2 \in \mathbb{R}\}$$

Visto que, cada módulo LOP **Spec** é um conjunto

$$\{S_1, \ldots, S_n, op_1, \ldots, op_m, ax_1, \ldots, ax_k\}$$

afirmamos que **Spec** é uma *classe* de objetos. Além disso, como **Spec** tem a forma

$$< \mathrm{Sig}_{Spec}, Ax >$$

onde $\mathrm{Sig}_{Spec} = <$ S, O $>$, e a cada módulo está associada uma teoria Th(**Spec**), os objetos associados a **Spec** são de duas naturezas: sintática e semântica. Os objetos sintáticos por sua vez são de três classes: os sorts, os operadores e os axiomas. Por outro lado, como mostra a figura Fig. 4.1, a cada classe LOP estão associadas duas classes de objetos semânticos: a primeira, formada pelos conjuntos carregadores, e a segunda, formada pelo conjunto de fórmulas associadas a **Spec** (teoria).

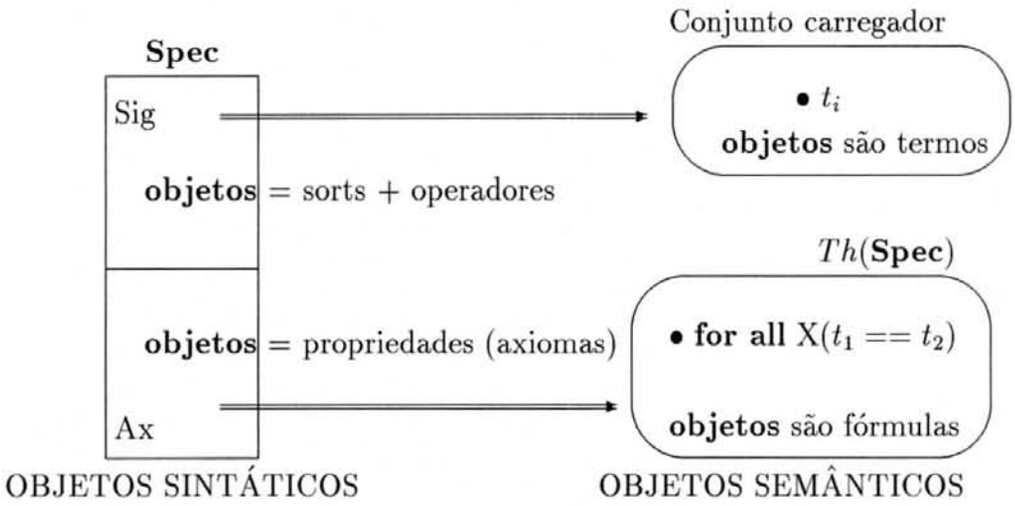


Figura 4.1 Objetos associados a uma classe LOP

As propriedades de cada classe LOP são dadas através da sua signatura e do conjunto de axiomas. No exemplo 2.2 da seção 2.2, a classe

$$\mathrm{QUEUE(Q,E)} = < \mathrm{Sig}_{QUEUE}, \{\mathrm{Q1, Q2, Q3, Q4}\} >$$

especifica um conjunto de objetos concretos chamados filas. Para esta classe existe uma classe de objetos Sig_{QUEUE}, formada pelo conjunto de sorts {Q, E, Bool}, outra formada pelo conjunto de operadores $\{newq, appendq, isemptyq, firstq, restq\}$ presentes na especificação, e uma terceira classe formada pelo conjunto de axiomas (ou propriedades) {Q1, Q2,Q3,Q4}. Associadas a estas classes existem ainda outras

duas classes implicitamente determinadas pela classe QUEUE(Q,E): uma formada pelos termos do sort principal QUEUE

$$\{newq(), ..., appendq(newq(), e), ... \}$$

e outra Th(QUEUE) formada pelas Sig_{QUEUE}-fórmulas. As classes QUEUE(Q,Integer) (filas de números inteiros), e QUEUE(Q,Array) (filas de array) também são classes semelhantes (filas específicas).

Uma observação importante que merece destaque nesta parte, refere-se à relação entre os tipos de classes que uma especificação LOP define quando esta é construída incrementalmente, isto é, quando uma classe esta incluída dentro de outra (classe-subclasse ou classe-superclasse).

A relação entre classes (concretas) implementada nas linguagens de programação consiste em relacionar duas classes de acordo com o número de seus elementos em proporção inversa ao número de atributos dos seus elementos: *o número de elementos de uma classe em um conjunto de classes concêntricas (umas includídas dentro de outras monotonicamente) é inversamente proporcional ao número de atributos que a define.*

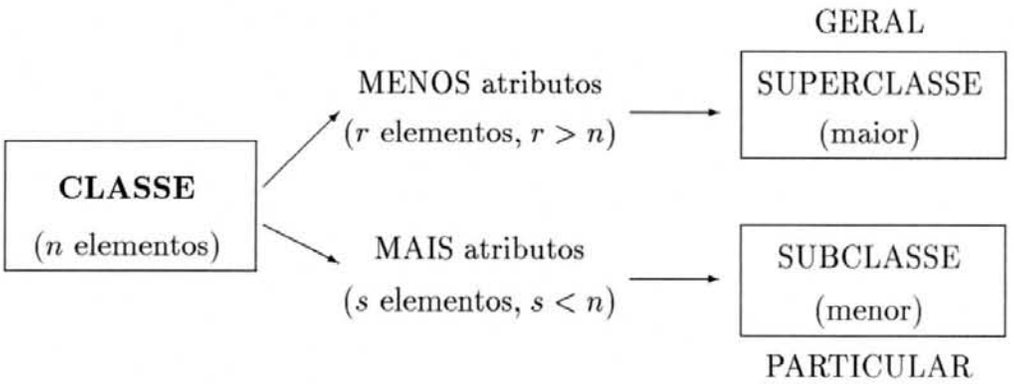


Figura 4.2 Classes (concretas) de acordo ao número de elementos

Visto que a linguagem LOP *não* é uma linguagem de programação e sim de especificação onde as classes são *abstratas*, os conceitos de classe tem sido adaptados sem contrariar o conceito acima que não pode ser aplicado neste caso, pois os objetos manipulados são de natureza abstrata onde os atributos dos objetos (concretos) são considerados também como objetos (operadores ou axiomas)

Duas classes LOP estão relacionadas unicamente através do mecanismo chamado de *extensão conservativa*: dadas duas classes S1 e S2, a classe S1 é uma subclasse de S2 (ou S2 é uma superclasse de S1), se os objetos de S1 formam um subconjunto dos objetos S2, denotado por S1 $\subseteq$ S2. Isto significa que uma subclasse em LOP sempre terá *menos* objetos (sintáticos e semânticos) que sua respectiva classe (superclasse) e viceversa. Assim, se S1 $\subseteq$ S2 então

conj. sorts de S1 $\subseteq$ conj. sorts de S2

conj. operadores de S1 $\subseteq$ conj. operadores de S2

conj. axiomas de S1 $\subseteq$ conj. axiomas de S2.

Vale a pena ressaltar aqui, que as classes LOP tem um conjunto finito de objetos sintáticos (sorts, operadores e axiomas).

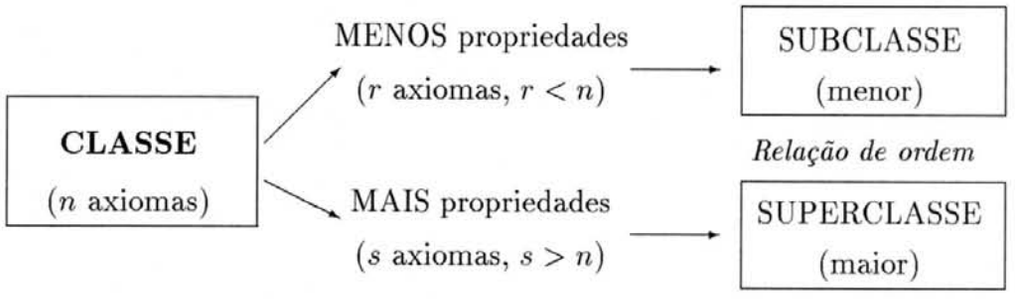


Figura 4.3 Classes (abstratas) de acordo a suas propriedades matemáticas

Desta forma podemos dizer que uma *superclasse* LOP é construída pela inclusão de sorts, operadores e/ou axiomas a outra classe base, e inversamente, uma

subclasse LOP é construída pela eliminação de sorts, operadores e/ou axiomas em outra classe base (ver subseção 4.2.1). Isto é mostrado no exemplo seguinte.

Exemplo 4.1

STACK1(C,E) : **class**

introduces

$new : \{\} \longrightarrow C$

$push : C, E \longrightarrow C$

asserts

C **generated by** *new*, *push*

STACK2(C,E) : **class**

introduces

$new : \{\} \longrightarrow C$

$push : C, E \longrightarrow C$

$pop : C \longrightarrow C$

$top : C \longrightarrow E$

asserts

C **generated by** *new*, *push*

for all s: C, e: E

$top(push(s, e)) == e$ $\quad - ST1 -$

$pop(push(s, e)) == e$ $\quad - ST2 -$

implies

converts *top*, *pop*

exempting $top(new())$, $pop(new())$

STACK3(C,E) : **class**

introduces

$$\begin{aligned} new &: \{\} \longrightarrow C \\ push &: C, E \longrightarrow C \\ pop &: C \longrightarrow C \\ top &: C \longrightarrow E \\ size &: C \longrightarrow Int \\ isempty &: C \longrightarrow Bool \end{aligned}$$

asserts

C **generated by** *new*, *push*

C **partitioned by** *top*, *size*, *pop*, *isempty*

for all s: C, e: E

$$top(push(s,e)) == e \qquad -ST1-$$

$$pop(push(s,e)) == e \qquad -ST2-$$

$$size(new()) == 0 \qquad -ST3-$$

$$size(push(s,e)) == 1 + size(s) \qquad -ST4-$$

$$isempty(new()) == true() \qquad -ST5-$$

$$isempty(push(s,e)) == false() \qquad -ST6-$$

implies

converts *top*, *pop*, *size*, *isempty*

exempting *top*(*new*()), *pop*(*new*())

Podemos re-escrever os módulos acima da seguinte maneira:

STACK1 = <Sig_{STACK1}, { } >

Sig_{STACK1} = < {C, E}, $\{new, push\}$ >

STACK2 = <Sig_{STACK2}, {ST1, ST2} >

Sig_{STACK2} = < {C, E}, $\{new, push, pop, top\}$ >

STACK3 = <Sig_{STACK3}, {ST1, ST2, ST3, ST4, ST5, ST6 } >

Sig_{STACK3} = < {C, E, Int, Bool}, $\{new, push, pop, top, size, isempty\}$ >

e verificar claramente que

$$STACK1 \subseteq STACK2 \subseteq STACK3$$

As classes (ou superclasses) STACK2 e STACK3 tem sido construídas em forma incremental a partir de STACK1 pela inclusão de sorts, operadores e axiomas. Poderia ter acontecido o contrário: a partir de STACK3 e pela eliminação de sorts, operadores e/ou axiomas, obtermos as subclasses STACK2 e STACK1.

Deve-se observar também aqui, que na linguagem LOP, não existe o conceito de *instância* (qualidade do que é instante, segundo o Dicionário Brasileiro da Língua Portuguesa), pois o elemento *tempo* é próprio das linguagens de programação e de programas sendo executados. Por exemplo, em OO Turbo Pascal, o objeto `temperatura` é definido como:

```
temperatura = object
              value : real;
              procedure PutTemp (T:real; DegType: char);
              function GetTempC : real;
              function GetTempF : real;
              end;
```

e as variáveis A,B e C são instâncias (cópias) de `Temperatura` que somente são criadas quando o procedimento `Calor` é executado.

```
procedure Calor;
    var
        A, B, C : Temperatura
        tc : real
    begin
        A.PutTemp(32,'F');
        tc : = B.GetTempC;
        ..................
```

Devemos salientar aqui novamente que, apesar do objetivo final de toda especificação formal é ser implementada em uma linguagem de programação (um programa a ser executado), o objetivo da linguagem LOP é somente fornecer um conjunto de símbolos e regras para *escrever formalmente* especificações que poderão ou não ser executáveis. Com isto queremos dizer, que as classes LOP não são instanciadas no estilo das linguagens de programação. O fato de todas as classes LOP serem potencialmente parametrizadas, significa que os parâmetros (nome da classe, sorts e/ou operadores) podem ser susbstituídos (renomeados) por outros símbolos (de sorts ou operadores) e a classe resultante continuará sendo parametrizada. Isto será considerado novamente nas próximas seções.

4.2 Herança em LOP

Em especificação algébrica, a construção incremental de especificações (umas a partir de outras) facilita a reusabilidade, modularidade, evolução, verificação redundante, etc., porém, implica na definição de mecanismos que expliquem como deve ser feito esta construção e qual é a relação entre duas especificações. Quando uma especificação (resultante) é construída sobre outra já existente (a base), o mecanismo que relaciona estas duas especificações, é conhecido como *herança.*

Para entender melhor as diferentes idéias existentes sobre herança, primeiro apresentaremos algumas definições importantes:

- [COA 92]: *Herança* é o mecanismo para expressar a similaridade entre classes, simplificando a definição de classes iguais a outras que já foram definidas. Ela representa generalização e especialização, tornando explícitos os Atributos e Serviços comuns em uma hierarquia de classe.

- [DAN 88]: *Herdar* é receber propriedades ou características de outro, normalmente como resultado de algum relacionamento entre o doador e o recevedor.

- [IER 91a]: *Herança* (de especificações) é um mecanismo para reusar especificações, permitindo ao projetista combinar e refinar especificações velhas para construir outras especificações novas.

- [RIN 92]: *Herança* é o relacionamento entre duas classes de modo que, uma das classes recebe todos os fatos relevantes da outra classe.

- [WEG 88]: *Herança* é uma classe particular do mecanismo de modificação incremental, que transforma uma entidade origem O com um modificador M em uma entidade resultante R.

  $$R = O + M$$

  Estas entidades são definidas por conjuntos de atributos.

- [WEG 92]: *Herança* é um mecanismo para compartilhar o código ou comportamento comum a uma coleção de classes. Herança estrutura classes. Herança classifica classes e classes classifica objetos.

Quanto à definição de *herança*, observamos existe um consensoem que a idéia principal é o relacionamento entre duas entidades (classes, módulos, etc.) sendo uma a base da outra. Na abordagem orientada a objetos, as entidades são classes (superclasse, classe base e subclasse), e em especificação algébrica são os módulos ou especificações e consequentemente os tipos de dados abstratos (TDAs). A diferença entre cada abordagem está na forma *como* é feito ou como é definido este relacionamento entre duas entidades, pois cada uma implementa um mecanismo particular de herança, baseados no desejo particular do *que* deve ser herdado.

Neste trabalho, motivado pelo consenso acima mencionado, e restringindo-nos somente aos aspectos abstratos da linguagem LOP, adotaremos a seguinte definição de **herança**:

> *Dadas duas classes* $Spec_A$ *e* $Spec_B$*, dizemos que a classe* $Spec_B$ **herda** *a classe* $Spec_A$ *se* $Spec_B$ *é uma extensão conservativa de* $Spec_A$*, isto é,*
>
> $Sig_A \subseteq Sig_B$
>
> $Ax_A \subseteq Ax_B$
>
> $Th(Spec_A) \subseteq Th(Spec_B)$
>
> $Spec_A$ é chamada de *classe base* ou *classe herdada* e $Spec_B$ é chamada de *superclasse* ou *classe modificada.*

Este tipo de herança é considerado como um caso particular do *mecanismo de modificação incremental* presente em sistemas físicos, matemáticos e de software, onde uma entidade A junto com um modificador M é transformado em outra entidade B

$$B = A + M,$$

onde $+$ é o operador composição e M (disjunto de A) é chamado *domínio de modificadores.*

Sejam

$\mathcal{A}$ = { $Spec_i$ / $Spec_i$ é uma classe LOP }
= conjunto de *classes base*

$\mathcal{B}$ = { $Spec_j$ / $Spec_j$ é uma classe LOP }
= conjunto de *superclasses* ou *classes modificadas*

Um subconjunto $\mathcal{H}$ do produto cartesiano $\mathcal{A} \times \mathcal{B}$ é **herança** (simples) quando $Spec_j$ é uma extensão conservativa de $Spec_i$.

Em cada relação de herança em LOP, isto é, quando uma classe $Spec_B$ herda uma classe $Spec_A$, utilizaremos a notação

$$Spec_B = Spec_A + M_A$$

onde M_A é o modificador da classe $Spec_A$ para obter a superclasse $Spec_B$.

Considerando a relação

$$\mathcal{H} = \{ (\mathrm{Spec}_A, \mathrm{Spec}_A + \mathrm{M}_A) / \ \mathrm{Spec}_A \text{ classe LOP } \}$$

podemos identificar a herança $\mathcal{H}$ pelo modificador M_A no sentido que **a classe Spec_B herda a classe Spec_A modificada por M_A**. Por simplicidade operacional, adotaremos a notação

$$\mathrm{her}(\mathrm{Spec}_A, \mathrm{Spec}_B) = \mathrm{M}_A$$

em substituição da notação clássica

$$\mathrm{Spec}_A \ \mathcal{H} \ \mathrm{Spec}_B$$

para os elementos de $\mathcal{H}$. Assim, a expressão

$$\mathrm{her}(\mathrm{Spec}_A, \mathrm{Spec}_B) = \mathrm{her}(\mathrm{Spec}_A, \mathrm{Spec}_A + \mathrm{M}_A) = \mathrm{M}_A$$

pode ser lida como: *a especificação Spec_B herda Spec_A modificada por M_A*. O domínio Mod para M_A em LOP, poderá ser um conjunto de sorts, operadores, axiomas, ou simplesmente, uma função de renomeação.

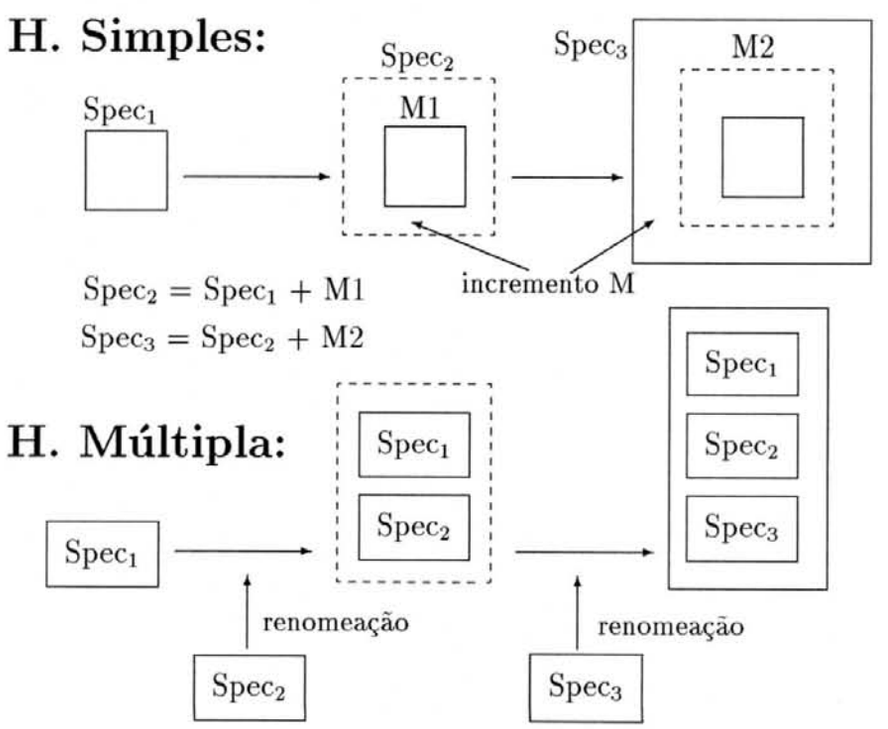


Figura 4.4 Construção Incremental em LOP

Como uma extensão do conceito anterior, dizemos que **Herança múltipla** é a relação

$$\mathcal{H} \subseteq \underbrace{\mathcal{A} \times \mathcal{A} \times \cdots \times \mathcal{A}}_{n-vezes} \times \mathcal{B}$$

com

$$her(\text{Spec}_1, \ldots, \text{Spec}_n, \text{Spec}) = \text{M}_n$$

entre n+1 classes, onde Spec_1, ..., $\text{Spec}_n \in \mathcal{A}$ são chamadas de *classes base* e Spec $\in \mathcal{B}$ é chamada de *classe modificada.*

4.2.1 Herança Estrutural e Herança Abstrata

Segundo [WEG 88], a natureza da herança considerada como um mecanismo de modificação incremental, é de dois tipos:

- *estrutural*, entre entidades cuja estrutura consiste de conjuntos finitos de atributos, e
- *abstrata*, onde entidades formam uma rede de herança com nós sem estrutura interna e onde as propriedades de herança são axiomas de ordem parcial.

Visto que, as características (sintáticas) de uma classe LOP são expressas através de sorts, operadores e axiomas, a herança estrutural (simples) na linguagem LOP é definida da seguinte forma. Sejam

$\text{Spec}_1 = <\text{Sig1}, \text{Ax1}>$ $\qquad$ $\text{Sig1} = < \text{S1}, \text{O1} >$

$\text{Spec}_2 = <\text{Sig2}, \text{Ax2}>$ $\qquad$ $\text{Sig2} = < \text{S2}, \text{O2} >$

então

$$
\begin{aligned}
her(Spec_1, Spec_2) &= her(<Sig1, Ax1 >, <Sig2, Ax2 >) \\
&= her(<Sig1, Ax1 >, Spec_1 + M >) \\
&= her(<Sig1, Ax1 >, <Sig1, Ax1 > + M) \\
&= her(<Sig1, Ax1 >, <Sig1 + Ms, Ax1+ Mx1 >) \\
&= her(<Sig1, Ax1 >, < <S1,O1> + Ms, Ax1+ Mx1 >) \\
&= her(<Sig1, Ax1 >, < <S1+Ms1,O1+Mo1>, Ax1+ Mx1 >) \\
&= her(<Sig1,Ax1>, <<S1,O1>,Ax1> + <<Ms1,Mo1>,Mx1>) \\
&= < <Ms1, Mo1>, Mx1>
\end{aligned}
$$

Observa-se que o modificador M é descomposto em três partes Ms1, Mo1, Mx1. Cada um destes representa a inclusão ou exclusão de sorts, operadores e axiomas, respectivamente, em relação a $Spec_1$. Se $Spec_2$ é o resultado da inclusão de sorts, operadores e axiomas em $Spec_1$, o modificador M é denotado por

$$M^+ = \{Ms1^+, Mo1^+, Mx1^+\}$$

e $Spec_2$ é chamada de *Superclasse* de $Spec_1$. Se pelo contrário, $Spec_2$ é o resultado da exclusão de sorts, operadores e axiomas em $Spec_1$, o modificador M é denotado por

$$M^- = \{Ms1^-, Mo1^-, Mx1^-\}$$

e $Spec_2$ é chamada de *subclasse* de $Spec_1$.

Analogamente, para o caso da herança estrutural múltipla, usando conceitos da teoria de conjuntos temos:

$$
\begin{aligned}
\text{her}(\text{Spec}_1, \ldots, \text{Spec}_n, \text{Spec}) &= \\
&= \text{her}(\text{Spec}_1,\ldots, \text{Spec}_n, <\text{Sig}, \text{Ax}>) \\
&= \text{her}(\text{Spec}_1,\ldots, \text{Spec}_n, \text{Spec}_1 \cup \ldots \cup \text{Spec}_n + \text{M}>) \\
&= \text{her}(\text{Spec}_1,\ldots, \text{Spec}_n, \textstyle\bigcup_{i=1}^{n} \text{Spec}_i + \text{M}) \\
&= \text{her}(\text{Spec}_1,\ldots, \text{Spec}_n, \textstyle\bigcup_{i=1}^{n} < \text{Sig}_i, \text{Ax}_i > + \text{M}) \\
&= \text{her}(\text{Spec}_1,\ldots, \text{Spec}_n, \textstyle\bigcup_{i=1}^{n} <\text{Sig}_i,\text{Ax}_i > + \\
&\qquad < \text{Sig} - \textstyle\bigcup_{i=1}^{n}\text{Sig}_i, \text{Ax} - \textstyle\bigcup_{i=1}^{n} \text{Ax}_i >) \\
&= < \text{Sig} - \textstyle\bigcup_{i=1}^{n}\text{Sig}_i, \text{Ax} - \textstyle\bigcup_{i=1}^{n} \text{Ax}_i > \\
&= < <\text{S},\text{O}> - \textstyle\bigcup_{i=1}^{n} < S_i, O_i >, \text{Ax} - \textstyle\bigcup_{i=1}^{n} \text{Ax}_i > \\
&= < <\text{S},\text{O}> - < \textstyle\bigcup_{i=1}^{n} S_i, \textstyle\bigcup_{i=1}^{n} O_i >, \text{Ax} - \textstyle\bigcup_{i=1}^{n} \text{Ax}_i > \\
&= < < \text{S} - \textstyle\bigcup_{i=1}^{n}\text{S}_i, \text{O} - \textstyle\bigcup_{i=1}^{n}\text{O}_i >, \text{Ax} - \textstyle\bigcup_{i=1}^{n}\text{Ax}_i >
\end{aligned}
$$

Assim,

$$\text{her}(\text{Spec}_1,\ldots,\text{Spec}_n,\text{Spec}) = < < \text{S} - \textstyle\bigcup_{i=1}^{n}\text{S}_i, \text{O} - \textstyle\bigcup_{i=1}^{n}\text{O}_i >, \text{Ax} - \textstyle\bigcup_{i=1}^{n}\text{Ax}_i >$$

A herança abstrata em LOP é definida através do relacionamento entre as teorias das especificações base e a teoria da especificação modificada. Para o caso de herança simples

$$\text{Th}(\text{Spec}_1)\ \mathcal{H}\ \text{Th}(\text{Spec}_2)$$

define-se como:

$$\text{her}(\text{Th}(\text{Spec}_1), \text{Th}(\text{Spec}_2)\ ) = \text{m}$$

onde $\text{m} = \{\ f \in \text{Th}(\text{Spec}_2)\ /\ f \notin \text{Th}(\text{Spec}_1)\ \}$ e $\text{Th}(\text{Spec}_1) \subseteq \text{Th}(\text{Spec}_2)$

4.3 Operações

Módulos de especificações são adaptados ou combinados através de várias operações usando mecanismos de renomeação (de sorts e operadores) e de importação. Uma operação deste tipo, deve ser preferivelmente simples e efecientemente computável sobre a representação textual dos módulos envolvidos na operação, e deve ter uma correspondente operação semântica, de modo que a com-

putabilidade entre os módulos seja praticável e significativa. Existem alguns estudos relacionados com operações entre módulos de especificações, como [BER 90], onde são abordados, por exemplo, álgebra de módulos, álgebra de signaturas, modelos de álgebras de módulos, etc. e como em OBJ3 onde são definidos expressões módulo, teorias e visões.

Estamos interessados agora em definir operações (abstratas) entre módulos (classes) LOP de modo que estas operações sejam compatíveis com aquelas conhecidas em Inteligência Artificial, Banco de Dados e Orientação a Objetos. As operações definidas entre classes LOP são:

- Especialização,
- Agregação, e
- Composição.

O conjunto de classes LOP doravante será denotado pelo sort *classLop*.

4.3.1 Gramática

specification	::=	class$^+$
class	::=	simpleId [({ name [: signature] }$^+$,)] : **class**
		{ shorthand | external }* opPart* propPart*
		[consequences]
external	::=	{ **specialization of** simpleId (renaming)
		| { aggregation | composition } }
aggregation	::=	**aggregation of** classRef$^+$
composition	::=	**composition of** classRef$^+$
classRef	::=	{ simpleId | (simpleId$^+$,) } [(renaming)]
renaming	::=	replace$^+$, | name$^+$, { , replace }*
replace	::=	name **for** name [: signature]

4.3.2 Especialização

Esta operação é aplicada sobre uma *classe parametrizada.* Entende-se por *parâmetros* de uma especificação, o conjunto de símbolos que podem ser substituídos por outros símbolos. Visto que, uma classe LOP pode ser escrita na forma:

$$Spec = << \{s_1, ..., s_n\}, \{op_1, ..., op_k\} >, Ax_1, \ldots, Ax_m >$$

então, todos os sorts s_i e todos os operadores op_j podem ser considerados como parâmetros potenciais de *Spec.* Estes parâmetros, para efeitos de utilização prática, podem ser classificados em genéricos (abstratos) e específicos (concretos), porém ambos com o mesmo significado, diferentemente das linguagens de programação onde os parâmetros são *formais* e *efetivos*, cada um com um significado diferente. Podemos considerar como parâmetros genéricos, por exemplo: C, E, f, Stack, Set, Item, Book, Database, Read, lookup, Fruta, get, init, etc. e como parâmetros específicos: Integer, StackOfWindows, BirthdayBook, Money, Pagamento, Fornecedor, Laranja, getSalario, closeFileDeposit, etc.. Os parâmetros podem ser *implícitos* se aparecem somente na signatura da classe, e *explícitos* se aparecem também como argumentos do nome da classe $NameSpec(par_1, ...par_n)$.

Uma especificação parametrizada S_1 é especializada em outra S_2 através da substituição de parâmetros genéricos em S_1 por parâmetros específicos em S_2 e é feita usando a cláusula

S_2 : **class**

specialization of S_1 (...**for**..., . . ., ...**for**...)

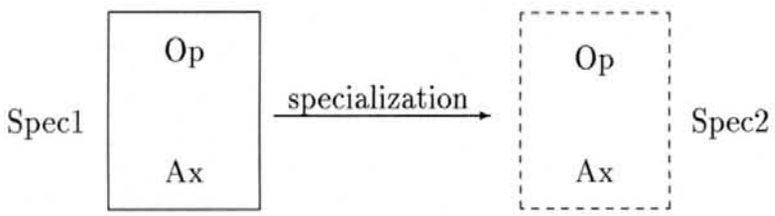


Figura 4.5 Operação de Especialização

Cada operação de especialização é definida como uma função

specialization of : classesLop, ParamSet $\longrightarrow$ classesLop

specialization of($< < S, O >, Ax >, < S^P, O^P >$) =

$= <$ **...for...**($< S, O >$, **...for...**(Ax)>

$= < < S^P, O^P >, Ax_{renamed} >$

onde a nova função **...for...** é chamada de *mecanismo de renomeação* e definida como:

...for... : Spec1 $\longrightarrow$ Spec2

...for... $(s_i) = s_i^p$ $\qquad s_i \in S,\ s_i^p \in S^P$

...for... $(o_j) = o_j^p$ $\qquad o_j \in O,\ o_j^p \in O^P$

...for... $\underbrace{(\forall X(t_1 == t_2))}_{ax_k} = \underbrace{\forall X(...\text{for}...(t_1) == ...\text{for}...(t_2))}_{ax_k^p}$

e

$$\begin{aligned} \textbf{...for...}(t(..., o_j, ...)) &= t(..., \textbf{...for...}(o_j), ...) \\ &= t(..., o_j^p, ...) \end{aligned}$$

onde t, t_1, t_2 são termos de qualquer axioma de Ax, Spec1 é a classe parametrizada e Spec2 é classe especializada. A função **...for...** poderá ser aplicada só para sorts, só para operadores e axiomas ou para ambos.

Para evitar conflitos na especificação resultante Spec2, por exemplo, dois operadores diferentes usando o mesmo símbolo (nome), a função **...for...** deve ser injetiva:

se $op_1 \neq op_2$ então **...for...**$(op_1) \neq$ **...for...**(op_2)

para qualquer par de operadores op_1 e op_2 em Spec1. Conseqüentemente,

se $t_1 \neq t_2$ então **...for...**$(t_1) \neq$ **...for...**(t_2)

para qualquer par de termos t_1, t_2 em Ax de Spec1.

Diferentemente das linguagens de programação, na linguagem LOP é possível estabelecer um processo infinito de especializações a partir de uma determinada classe parametrizada, no sentido de que *toda classe especializada é novamente uma classe parametrizada*:

S_2 : **class**

specialization of S_1 (a **for** A, b **for** B, c **for** C)

S_3 : **class**

specialization of S_2 (x **for** A, y **for** B, z **for** C)

S_4 : **class**

specialization of S_3 (U **for** x, V **for** y, W **for** z)

.....

.....

Assim, por exemplo, a partir de STACK3(C,E) (definida na seção 4.1), podemos ter

$STACK\_INT$: **class**

specialization of $STACK$ (StackInt **for** C, Int **for** E)

$STACK\_NAT$: **class**

specialization of $STACK\_INT$ (StackNat **for** StackInt, Nat **for** Int)

$STACK\_VECT$: **class**

specialization of $STACK\_NAT$ (StackVect **for** StackNat, Vect **for** Nat)

.....

Na operação de especialização, a classe Spec2 *herda* todos os sorts, operadores e axiomas renomeados em Spec1. Neste caso, o modificador é a função de renomeação **...for...**, i.é.,

her(Spec1, Spec2) = Spec1 + **...for...**

Isto chamamos de *herança horizontal.* Além disso, podemos estabelecer um homo-

morfismo h entre as duas teorias associadas

$$\text{h} : \text{Th(Spec1)} \longrightarrow \text{Th(Spec2)}$$

definido como

$$\text{h}(\forall X(t_1 == t_2)\ ) = \forall X(\textbf{...for...}(t_1) == \textbf{...for...}(t_2)\ )$$

e mostrar que h é bijetivo. Em efeito,

suponha-se que $\forall X(t_1 == t_2) \neq \forall Y(r_1 == r_2)$ em Spec1. É obvio que $t_1 \neq r_1$ e $t_2 \neq r_2$. Logo, **...for...**$(t_1) \neq$ **...for...**(r_1) e também **...for...**$(t_2) \neq$ **...for...**(r_2)

$\Rightarrow\ \forall X(\textbf{...for...}(t_1) == \textbf{...for...}(t_2)\ ) \neq \forall Y(\textbf{...for...}(r_1) == \textbf{...for...}(r_2)\ )$

$\Rightarrow$ h$(\forall X(t_1 == t_2)\ ) \neq h(\forall Y(r_1 == r_2))$. Logo h é injetivo. Como Spec2 é uma classe especializada (renomeada) a partir de Spec1, cada axioma em Th(Spec2) é da forma **...for...**$(\forall X(t_1 == t_2)\ )$. Em particular, se $\forall Y(r_1 == r_2) \in$ Th(Spec2) então existe $\forall X(t_1 == t_2)$ em Th(Spec1) tal que

$$\text{h}(\forall X(t_1 == t_2)\ ) = \forall X(r_1 == r_2)$$

logo, h também e surjetivo. Sendo injetivo e surjetivo, h é isomorfismo, de modo que a teoria Th(Spec2) resulta ser uma imagem isomorfica da teoria Th(Spec1):

$$\text{Th(Spec1)} \sim \text{Th(Spec2)}$$

Exemplo 4.2

Uma parte da especificação de um transmissor com três portas (leitura, transmissão e controle) em um protocolo de comunicação é mostrado a seguir. Os operadores composição sequencial $\odot$, composição alternativa $\oplus$, *read* e *send* são definidos no próximo capítulo.

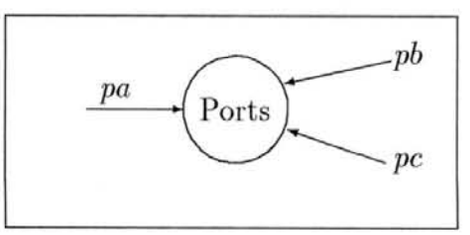
PORTS(pa, pb, pc) : **class**

introduces

pa : { } $\longrightarrow$ Ports

pb : { } $\longrightarrow$ Ports

pc : { } $\longrightarrow$ Ports



asserts

Ports **generated by** *pa, pb, pc*

TRANSMITTER(pa, pb, pc) : **class**

aggregation of ALG_PROCESS, INTERACTION, MESSAGES, PORTS

introduces

transmitter	:	{ }	$\longrightarrow$	Process
read_host	:	Boolean	$\longrightarrow$	Process
send_frame	:	Data, Boolean	$\longrightarrow$	Process
wait_send	:	Message	$\longrightarrow$	Process

asserts

for all b: Boolean, d: Data

$$read\_host(b) == read(pa, frame(d, true())) \odot send\_frame(d, b) \qquad TR1$$

$$send\_frame(d, b) == send(pb, frame(d, b)) \odot wait\_send(frame(d, b)) \qquad TR2$$

$$wait\_send(frame(d, b)) == read(pc, k) \odot read\_host(\neg b) \oplus$$
$$((read(pc, ch\_error()) \oplus time\_out()) \odot send\_frame(d, b)) \qquad TR3$$

$$transmitter() == read\_host(false()) \qquad TR4$$

Esta especificação pode ser especializada para especificar um particular protocolo TRC (transmissor com portas p1, p3 e p4) da seguinte forma:

T_TRC : **class**

specialization of TRANSMITTER(p1 **for** pa, p3 **for** pb, p4 **for** pc)

ou equivalentemente

T_TRC : **class**

specialization of TRANSMITTER(p1, p3, p4)

No exemplo, a especificação T_TRC herda todos os sorts, operadores e axiomas definidos em TRANSMITTER, onde somente foram renomeados os operadores pa, pb e pc. Os axiomas da classe T_TRC são:

asserts

for all b: Boolean, d: Data

$$read\_host(b) == read(p1, frame(d, true())) \odot send\_frame(d, b) \qquad TR1'$$

$$send\_frame(d,b) == send(p3, frame(d,b)) \odot wait\_send(frame(d,b)) \qquad TR2'$$

$$wait\_send(frame(d,b)) == read(p4,k) \odot read\_host(\neg b) \oplus$$

$$((read(p4, ch\_error()) \oplus time\_out()) \odot send\_frame(d,b)) \qquad TR3'$$

$$transmitter() == read\_host(false()) \qquad TR4'$$

É fácil verificar que:

$$\{TR1, TR2, TR3, TR4\} \sim \{TR1', TR2', TR3', TR4'\}.$$

4.3.3 Agregação

Especificações LOP são construídas incrementalmente: umas a partir de outras, de tal forma que, um módulo S pode ser considerado como um agregado de outros módulos pré-existentes $Spec_1, ..., Spec_n$.

A operação de agregação é implementada usando a cláusula

S : **class**

aggregation of $Spec_1$, ..., $Spec_n$

introduces

. . . . . .

asserts . . . . . .

e é definida como:

aggregation of : classLop, . . . , classLop $\longrightarrow$ classLop

aggregation of$(Spec_1, ..., Spec_n)$ =

= **aggregation of**$(< < S_1, O_1 >, Ax_1 >, ... , < < S_n, O_n >, Ax_n >)$

$= < < \cup_{i=1}^{n} S_i, \cup_{j=1}^{n} O_j >, \cup_{k=1}^{n} Ax_k >$.

Os conjuntos $S_i \cap S_j$, $O_i \cap O_j$ e $Ax_i \cap Ax_j$ devem ser vazios para cada $i \neq j$, caso contrário, deve ser aplicado antes o mecanismo de renomeção ...**for**... para sorts e operadores evitando desta forma inconsistências.

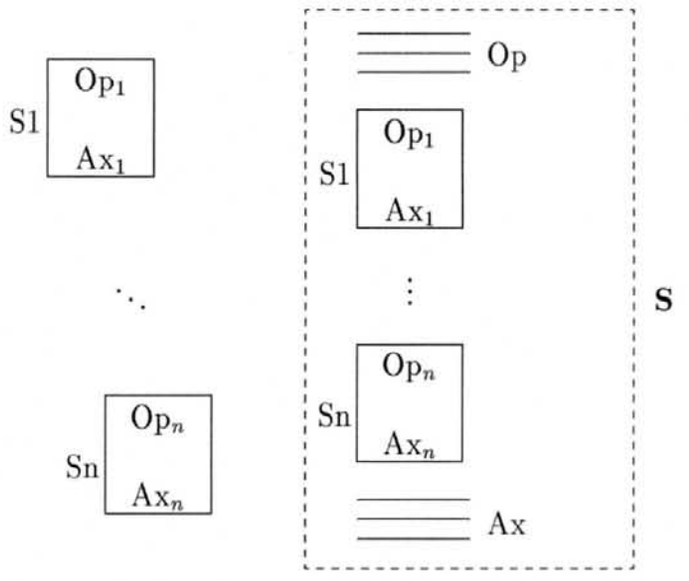


Figura 4.6 Operação de Agregação

Se Spec = < <S, Op>, Ax>, então a união dos conjuntos de operadores de todos os módulos $Spec_1, ..., Spec_n$, é um subconjunto de Op e também a união dos conjuntos de axiomas de todos os módulos $Spec_1, ..., Spec_n$, é um subconjunto de Ax.

Spec = $< < S_{Spec} \cup S_1 \cup ... \cup S_n, O_{Spec} \cup O_1 \cup ... \cup O_n >, \ Ax_{Spec} \cup Ax_1 \cup ... \cup Ax_n >$

Desta forma a teoria $Th(Spec)$ é uma extensão conservativa de cada teoria $Th(Spec_i)$, $i = 1, n$ e também

$$\bigcup_{i=1}^{n} Th(Spec_i) \quad \subseteq Th(Spec)$$

Exemplo 4.3

Supõe-se definido as classes BPA (operações básicas para processos) e DRINK (onde são definidos os tipos de bebida, preços e os processos de seleção de bebida) e deseja-se especificar uma pessoa tomando sua bebida favorita. Então usando a operação de agregação sobre as duas classes, mais a definição de dois novos operadores e um axioma, obtemos a seguinte classe:

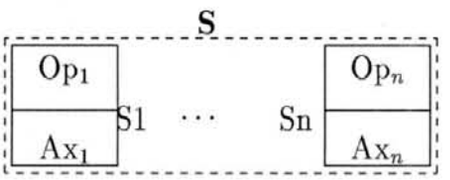


Figura 4.7 Operação de Composição

DRINKS_USER : **class**

aggregation of BPA, DRINKS

introduces

$user$: Drink $\longrightarrow$ Process

fav_drink : { } $\longrightarrow$ Drink

asserts

$user(fav\_drink()) == select(fav\_drink()) \odot$ $-DRU-$

$insert(price(fav\_drink())) \odot take\_drink(fav\_drink())$

Então,

$$\text{DRINK\_USER} = << S_{du}, O_{du} >, Ax_{du} >$$

onde

S_{du} = {Drink, Process}$\cup S_{BPA} \cup S_{DRINKS}$,

$O_{du} = \{user, fav\_drink\} \cup O_{BPA} \cup O_{DRINKS}$

Ax_{du} = {DRU}$\cup Ax_{BPA} \cup Ax_{DRINKS}$

4.3.4 Composição

Esta operação é um caso particular da operação de agregação e é implementada usando a cláusula:

Spec : **class**

composition of $Spec_1$, ..., $Spec_n$

$$\text{Spec} = << S_1 \cup ... \cup S_n, O_1 \cup ... \cup O_n >,\ Ax_1 \cup ... \cup Ax_n >$$

Quando uma especificação *Spec* é a composição das especificações $Spec_1, \ldots, Spec_n$, a união das teorias de todas as especificações é exatamente igual à teoria de *Spec*:

$$\bigcup_{i=1}^{n} Th(Spec_i) = Th(Spec)$$

Igual que na operação de agregação, primeiro devem ser removidas as inconsistências de sorts e operadores iguais.

Exemplo. Uma relação de equivalência é especificada usando a operação de composição, da seguinte forma:

EQUIVALENCE(eq, T) : **class**

composition of REFLEXIVE(eq,T), SYMMETRIC(eq,T),
TRANSITIVE(eq,T)

5 PROCESSOS

Durante muitos anos, a pesquisa teórica sobre processos ficou centralizada sobre as Álgebras de Processos: a área da ciência da computação teórica onde os sistemas concorrentes são modelados como elementos de uma álgebra. Desde o Calculus of Communicating Systems (CCS) de Milner [MIL 80] até a Álgebra de Processos Comunicantes (ACP) de Bergstra e Klop [BER 89], muitas outras álgebras de processos foram definidas e quase todas relacionadas umas com outras. Pela sua natureza teórica, somente nestes últimos cinco anos, estas álgebras foram usadas para especificar alguns pequenos sistemas concorrentes.

Especificação algébrica de sistemas concorrentes é a especificação de estruturas onde alguns dados (componentes) são processos ou estados de processos. Informalmente, *processos* são entidades com a capacidade de realizar uma atividade (conjunto de eventos), interagir com outras entidades e/ou seu ambiente através de comunicação sincronizada, paralelismo etc. ([AST 93]). Un *sistema concorrente* pode ser entendido como um processo, onde os componentes são também processos que operam concorrentemente.

Desde que R. Milner [MIL 80], [MIL 83] apresentou seu *Calculus of Communicating Systems* (CCS) no inicio da década passada, muitos trabalhos tem-se desenvolvido ao redor da modelagem de sistemas concorrentes como elementos de uma álgebra. A maioria destes trabalhos pertencem ao Grupo de Amsterdam e principalmente a J. A. Bergstra [BER 84], [BER 89] e J.C.M. Baeten [BAE 90], [BAE 91], [BAE 92], sob o título de *Álgebra de Processos*

Referências bibliográficas sobre especificação de sistemas concorrentes apontam a vários tipos de álgebras de processos de modo que as pessoas ficam um pouco confusas quando tentam usa-las, perguntando-se por exemplo, qual é a *última* álgebra de processos?. Segundo [van 89], *deve-se estar inteirado do fato de*

que existe um estreito relacionamento entre as muitas álgebras de processos: as vezes uma álgebra de processos pode ser vista como uma imagem homomórfica, como uma subálgebra ou como a restrição de outra álgebra.

Por outro lado, a existência de muitos trabalhos sobre concorrência e especificação algébrica, levam a pensar que em breve existirá uma proliferação de linguagens de especificação formal que aumentarão a confusão, principalmente dos iniciantes no estudo das álgebras de processos.

Neste capítulo, trataremos de incorporar à linguagem LOP, a maioria dos operadores das álgebras de processos existentes, na forma de sistemas de axiomas (especificações) e através de uma biblioteca de módulos, que chamaremos de ALG_PROCESS. Esta biblioteca é composta de módulos básicos e módulos avançados, que são construídos seguindo o método de construção incremental de especificações.

Observe-se que é mantida o uso da palavra 'álgebra' para mostrar a compatibilidade com as definições originais das referências bibliograficas.

5.1 Sistemas de Axiomas para Especificar Processos

A especificação de processos é abstraída a dois sorts principais e às operações sobre estes: *Atoms* e *Process*. O primeiro representa o conjunto de "steps" ou ações (eventos). Estas são atômicas e instantâneas. O sort *Process* representa o conjunto de processos. Um *processo* formalmente define-se como uma seqüência (conjunto indexado) de ações: $p = < a_1, \ldots, a_n >$ (finito) ou $p = < a_1, \ldots, a_n, \ldots >$ (infinito)

Um aspecto importante nesta abordagem é o fato de que os *processos* são modelados como elementos de uma Álgebra. Existem outras abordagens onde

processos são modelados como dados dinâmicos (sistemas de transição rotulados), estruturas de eventos, teorias temporais, conjuntos parcialmente ordenados, Redes de Petri, etc.

5.1.1 Álgebra de Processos Básica - BPA

O ponto de partida é um sistema de axiomas simples para especificar processos, chamado **Álgebra de Processos Básica** (BPA, Basic Process Algebra). Este sistema será o núcleo para a especificação de outros sistemas de axiomas necessários na especificação de processos. A signatura de BPA é definida por

$$Sig(BPA) =< \{Atoms, Process\}, \{i, \oplus, \odot\} >$$

O operador binário $\oplus$ é chamado de *composição alternativa* ou soma. O operador $\odot$ é chamado de *composição seqüencial*, precedência ou produto.[1] Para ter um tratamento uniforme entre ações e processos, todas as ações são consideradas processos através do operador *injeção* i

BPA : **class**

introduces

$i\_$: Atoms $\longrightarrow$ Process

$\_\oplus\_$: Process, Process $\longrightarrow$ Process

$\_\odot\_$: Process, Process $\longrightarrow$ Process

asserts

Process **generated by** $i\_$, $\oplus$, $\odot$

for all x, y, z: Process

$x \oplus y == y \oplus x$ $A1$

$x \oplus (y \oplus z) == (x \oplus y) \oplus z$ $A2$

[1] Em quase toda a literatura sobre processos são usados os símbolos "+" e "·" para representar a composição alternativa e composição seqüencial, respectivamente. Usamos os símbolos $\oplus$ e $\odot$ para evitar confusão com os símbolos da soma e produto de números naturais ou inteiros que poderão ser importados através de algum módulo em LOP.

$$x \oplus x == x \qquad A3$$

$$(x \oplus y) \odot z == x \odot z \ \oplus \ y \odot z \qquad A4$$

$$(x \odot y) \odot z == x \odot (y \odot z) \qquad A5$$

Pela signatura de BPA, observamos que qualquer processo pode ser especificado usando os operadores construtores i, $\oplus$ e $\odot$, isto é, qualquer processo é especificado na forma de i_a, $x \oplus y$ ou $x \odot y$. A interpretação da expressão $(i_a \oplus i_b) \odot i_c$ é um processo que, primeiro executa ou i_a ou i_b, e imediatamente depois executa o processo i_c e logo termina.

É importante observar, na especificação BPA, a ausência da lei distributiva a esquerda

$$x \odot (y \oplus z) == x \odot y \ \oplus \ x \odot z$$

para o operador $\odot$, que será explicada mais adiante.

5.1.2 Outros Operadores sobre Processos

- *Paralisação*

  O operador δ gera o processo constante $\delta()$. Este denota o conhecimento de paralisação, isto é, a constante $\delta()$ pode ser caracterizada operacionalmente como um processo incapaz de fazer alguma coisa ou executar alguma ação. O processo δ pode ser considerado como uma forma de terminação sem sucesso, isto é, δ pode ser tratado como a última parte (último evento) de um processo que não termina. O operador δ corresponde a NIL em CCS, STOP em TCSP e LOTOS. A especificação deste operador é a seguinte:

DEADLOCK : **class**

aggregation of BPA

introduces

$\delta : \{\ \} \longrightarrow$ Process

asserts

for all x: Process

$\delta() \oplus x == x$ *A6*

$\delta() \odot x == \delta()$ A7

A especificação BPA junto com a apresentação (signatura + axiomas) do operador δ é chamado de BPA_δ.

Usando o processo δ (axioma A6) podemos verificar agora a existência da segunda lei distributiva:

$$\begin{aligned} y \odot x &= y \odot (x + \delta()\ ) \\ &= y \odot x + y \odot \delta() \end{aligned}$$

Segundo esta igualdade, um processo *sem* possibilidade de paralisação (lado esquerdo) é igual a outro processo *com* possibilidade de paralisação (lado direito), o que representa uma contradição. Logo a lei $x \odot (y + z) = x \odot y + x \odot z$ (distributiva a esquerda) *não vale* em uma álgebra de processos onde é definido o processo δ. Isto é decorrente da ausência natural do axioma comutativo

$$x \odot y == y \odot x$$

- *Concorrência*

A concorrência entre dois processos é especificada através do operador $\|$, chamado de *composição paralela* ou concorrência. Se $x, y \in$ Process, então o termo $x \| y$, a execução paralela dos processos x e y, representa o processo que primeiro executa uma ação inicial a de x ou y e imediatamente depois executa a composição paralela dos restos $x - \{a\}$ e $y - \{a\}$ de x e y respectivamente [2], isto é, se $a \in x$ ou $a \in y$ então:

[2] O termo $x - \{a\}$ entenda-se como *o conjunto x sem o elemento a*

$$x \| y = a \odot ((x - \{a\}) \| (y - \{a\})$$

Isto significa que, as ações de x ou y são executadas em forma entrelaçada. A composição paralela não é definida diretamente, mas usando o operador auxiliar Merge Livre $\lfloor\!\lfloor$. A interpretação de $\lfloor\!\lfloor$ é similar à de $\|$. O termo $x \lfloor\!\lfloor y$ é interpretado como a execução paralela dos processos x e y, onde a ação inicial a pertence ao processo x:

$$x \lfloor\!\lfloor y = a \odot ((x - \{a\}) \| y), \quad a \in x$$

CONCURR : **class**

aggregation of BPA

introduces

$\_ \| \_$: Process, Process $\longrightarrow$ Process

$\_ \lfloor\!\lfloor \_$: Process, Process $\longrightarrow$ Process

asserts

for all a: Atoms, x, y, z: Process

$x \| y == (x \lfloor\!\lfloor y) \oplus (y \lfloor\!\lfloor x)$ $M1$

$i_a \lfloor\!\lfloor x == i_a \odot x$ $M2$

$(i_a \odot x) \lfloor\!\lfloor y == i_a \odot (x \| y)$ $M3$

$(x \oplus y) \lfloor\!\lfloor z == (x \lfloor\!\lfloor z) \oplus (y \lfloor\!\lfloor z)$ $M4$

Um fator importante a ser considerado aqui em relação ao processo $x \| y$, é o fato que o operador $\|$ não envolve comunicação alguma entre os dois processos: as ações dos processos x e y são *livremente entrelaçadas.*

- *Comunicação Binária*

Outro operador importante nas Álgebras de Processos é a comunicação binária $|$ entre a ação a de um processo e uma ação b de outro processo.

BIN_COMM : **class**

aggregation of DEADLOCK

introduces

$\_ | \_$: Process, Process $\longrightarrow$ Process

asserts

for all a, b, c: Atoms

$i_a|i_b == i_b|i_a$ $\quad C1$

$(i_a|i_b)|i_c == i_a|(i_b|i_c)$ $\quad C2$

$\delta()|i_a == \delta()$ $\quad C3$

Observa-se que o operador $|$ é comutativo, associativo e tem o operador δ como elemento zero. Se $i_a|i_b \neq \delta()$, significa então que, a e b podem sincronizar e neste caso, a e b são chamadas *ações de comunicação* ou ações subatômicas.

- *Merge com Comunicação*

  Usando o operador $|$, definido na classe BIN_COMM, agora podemos dar outra definição para os operadores $\|$ e $\lfloor\!\|$, apresentados na classe CONCURR. Esta nova classe, chamada de MERGE_COMM, é usada para definir posteriormente o sistema ACP para processos comunicantes, i.e., onde existe comunicação entre as ações de um processo x e as ações de outro processo y. A interpretação de $\lfloor\!\|$ é igual à do módulo CONCURR: $x\lfloor\!\| y = a \odot (\ (x - \{a\}) \ \| \ y), \quad a \in x$

  MERGE_COMM : **class**

  aggregation of BIN_COMM

  introduces

  $\_ \| \_$: Process, Process $\longrightarrow$ Process

  $\_ \lfloor\!\| \_$: Process, Process $\longrightarrow$ Process

  asserts

  for all a, b: Atoms, x, y, z: Process

  $x\|y == (x\lfloor\!\| y) \oplus (y\lfloor\!\| x) \oplus (x|y)$ $\quad CM1$

  $(i_a \odot x)\lfloor\!\| y == i_a \odot (x\lfloor\!\| y)$ $\quad CM2$

  $i_a\lfloor\!\| x == i_a \odot x$ $\quad CM3$

  $(x \oplus y)\lfloor\!\| z == (x\lfloor\!\| z) \oplus (y\lfloor\!\| z)$ $\quad CM4$

  $(i_a \odot x)|i_b == (i_a|i_b) \odot x$ $\quad CM5$

  $i_a|(i_b \odot x) == (i_a|i_b) \odot x$ $\quad CM6$

$$(i_a \odot x)|(i_b \odot y) == (i_a|i_b)(x \| y) \qquad CM7$$

$$(x \oplus y)|z == x|z \oplus y|z \qquad CM8$$

$$x|(y \oplus z) == x|y \oplus x|z \qquad CM9$$

Por outro lado, se para qualquer $a, b \in Atoms$, temos $i_a|i_b = \delta()$ (as ações atômicas a e b não podem se comunicar), então podemos dizer que *merge livre* é uma instância de *merge com comunicação*

- *Encapsulamento*

Algumas vezes não é possível a comunicação entre certas ações de dois processos. Para controlar quais ações podem e quais não podem se comunicar, é usado o operador de encapsulamento ∂. Este toma como entrada um processo e um conjunto s de ações que serão encapsuladas. O conjunto s representa todas as tentativas de comunicação sem sucesso. Cada subprocesso i_a de x, onde a é uma ação do conjunto s, será substituído pelo processo $\delta()$, de modo que o processo x será forçado a fazer uma escolha alternativa quando esta é possível. O operador de encapsulamento ∂ será usado para construir o sistema ACP (para processos comunicantes). O operador $\in$ é definido no módulo SET.

ENCAPSULATION(s) : **class**

aggregation of DEADLOCK, SET(C, Atoms)

introduces

∂ : Set, Process $\longrightarrow$ Process

asserts

for all a: Atoms, x, y: Process, s: C

$\partial(s, i_a) ==$ **if** $a \in s$)

then $\delta()$

else i_a $\qquad D1$

$$\partial(s, x \oplus y) == \partial(s, x) \oplus \partial(s, y) \qquad D2$$

$$\partial(s, x \odot y) == \partial(s, x) \odot \partial(s, y) \qquad D3$$

Tipicamente, um sistema de processos comunicantes $x_1, \ldots, x_n$ é representado pelas expressões:

$$\partial(s, x_1 \| \cdots \| x_n) \quad \text{ou} \quad \partial_s(x_1 \| \cdots \| x_n)$$

Prefixando o operador de encapsulamento, dizemos que o sistema $x_1, \ldots, x_n$ será considerado como uma unidade separada em relação às ações de comunicação mencionadas no conjunto s: nenhuma comunicação entre as ações de s e seu ambiente será esperado.

- *Concorrência Padrão*

STANDARD_CONC : **class**

aggregation of MERGE_COM

asserts

for all a: Atoms, x, y, z: Process

$$(x \mathbin{\|\!\_} y) \mathbin{\|\!\_} z == x \mathbin{\|\!\_} (y \| z) \qquad SC1$$

$$(x | (i_a \odot y)) \mathbin{\|\!\_} z == x | ((i_a \odot y) \mathbin{\|\!\_} z) \qquad SC2$$

$$x | y == y | x \qquad SC3$$

$$x \| y == y \| x \qquad SC4$$

$$x | (y | z) == (x | y) | z \qquad SC5$$

$$x \| (y \| z) == (x \| y) \| z \qquad SC6$$

- *Abstração*

Um aspecto fundamental no projeto e especificação de sistemas hierárquicos de processos comunicantes é a *abstração*. Sem um mecanismo de abstração que permita abstrair os eventos internos dos módulos que serão usados para construir sistemas grandes, a especificação poderia ser quase impossível. Em [BER 89], é apresentado um exemplo da necessidade de usar o mecanismo de abstração para especificar o comportamento do sistema composto B_{13}, como mostrado na fig. 5.1.

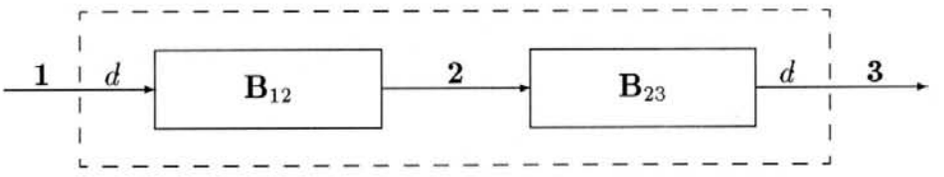


Figura 5.1 Bag Transparente $\mathbf{B}_{13}$

Neste exemplo, o único interesse é a passagem do dado d pelas portas 1 e 3, isto é, o comportamento do canal tracejado. Para abstrair os eventos que ocorrem nos canais B_{12} e B_{23}, precisa-se definir um operador que esconda todos os eventos internos destes dois canais: os eventos de comunicação. Pensando nisto define-se um operador de abstração *abst* com entrada um processo e um conjunto s de ações. O termo $abst(s, x)$ é denotado também por $abst_s(x)$. O operador τ é o operador do CCS denotando uma ação 'silenciosa'.

ABSTRACTION(s) : **class**

aggregation of BPA, SET(C, Atoms)

introduces

τ : {} $\longrightarrow$ Atoms

abst : Set, Process $\longrightarrow$ Process

asserts

for all a: Atoms, x, y: Process, s: C

$abst(s, i_\tau) == i_\tau$ $\quad TS1$

$abst(s, i_a) ==$ **if** $a \in s$

then i_τ

else i_a $\quad TS2$

$abst(s, x \oplus y) == abst(s, x) \oplus abst(s, y)$ $\quad TS3$

$abst(s, x \odot y) == abst(s, x) \odot abst(s, y)$ $\quad TS4$

- *Projeção*

  Para um determinado $n \in$N, o operador π *paralisa* um processo após este ter executado n ações atômicas, com a observação que, os τ-eventos são transparentes: $\pi(n, i_\tau) == i_\tau$. Este módulo é parametrizado pelo número natural n.

  PROJECTION(n) : **class**

  aggregation of ABSTRACTION, NATURAL

  introduces

  π : Nat, Process $\longrightarrow$ Process

  asserts

  for all a: Atoms, x, y: Process

  $\pi(n, i_\tau) == i_\tau$ $PR1$

  $\pi(zero(), i_a \odot x) == \delta()$ $PR2$

  $\pi(succ(n), i_a \odot x) == i_a \odot \pi(n, x)$ $PR3$

  $\pi(n, i_\tau \odot x) == i_\tau \pi(n, x)$ $PR4$

  $\pi(n, x \oplus y) == \pi(n, x) \oplus \pi(n, y)$ $PR5$

- *Renomeação*

  Este módulo foi definido em [van 89] e permite que encapsulamento e abstração sejam considerados como casos particulares de renomeação.

  RENAMINGS : **class**

  aggregation of ABSTRACTION

  introduces

  f, g : Process $\longrightarrow$ Process

  id : Process $\longrightarrow$ Process

  ρ : Process $\longrightarrow$ Process

  asserts

  for all x, y: Process

  $f(\delta()) == \delta()$ $RN1$

$$f(i_\tau) == i_\tau \qquad RN2$$

$$id(x) == x \qquad RN3$$

$$\rho(f(i_a)) == f(i_a) \qquad RN4$$

$$\rho(f(x \oplus y)) == \rho(f(x)) \oplus \rho(f(y)) \qquad RN5$$

$$\rho(f(x \odot y)) == \rho(f(x)) \odot \rho(f(y)) \qquad RN6$$

$$\rho(id(x)) == x \qquad RN7$$

$$\rho(f) \circ \rho(g)(x) == \rho(f \circ g(x)) \qquad RN8$$

- *"Silent"*

SILENT_STEP : **class**

aggregation of ABSTRACTION(I)

asserts

for all a: Atoms, x, y: Process

$$x \odot i_\tau == x \qquad T1$$

$$i_\tau \odot x \oplus x == i_\tau \odot x \qquad T2$$

$$i_a \odot (i_\tau \odot x \oplus y) == i_a \odot (i_\tau \odot x \oplus y) \oplus (i_a \odot x) \qquad T3$$

- *Silent-Merge*

SIL_MER : **class**

aggregation of SILENT_STEP, MERGE_COMM

asserts

for all x, y: Process

$$i_\tau \between x == i_\tau \odot x \qquad TM1$$

$$(i_\tau \odot x) \between y == i_\tau \odot (x \| y) \qquad TM2$$

$$i_\tau | x == \delta() \qquad TC1$$

$$x | i_\tau == \delta() \qquad TC2$$

$$(i_\tau \odot x) | y == x | y \qquad TC3$$

$$x | (i_\tau \odot y) == x | y \qquad TC4$$

- *Regiões Compactas de um Processo*

  Em processos comunicantes, *regiões compactas* (tight regions) são seqüências de ações (eventos) onde os processos são executados sem qualquer interrupção. O operador *tight* permite implementar este tipo de regiões. Intuitivamente, $tight(x)$ é o conjunto de eventos que serão executados em uma cooperação (execução paralela) como um simples evento atômico, isto é, nenhuma interrupção por parte de uma ação de outro processo paralelo é possível. O operador *tight* será usado para definir os módulos AMP e AMP(:). O operador ϕ remove as restrições sobre regiões compactas.

TIGHT : **class**

aggregation of BPA

introduces

$tight$: Process $\longrightarrow$ Process

ϕ: Process $\longrightarrow$ Process

asserts

for all a: Atoms, x, y: Process

$tight(i_a) == i_a$ $TR1$

$tight(x \oplus y) == tight(x) \oplus tight(y)$ $TR2$

$tight(tight(x)) == tight(x)$ $TR3$

$\phi(i_a) == i_a$ $F1$

$\phi(x \oplus y) == \phi(x) \oplus \phi(y)$ $F2$

$\phi(tight(x)) == \phi(x)$ $F3$

$\phi(x \odot y) == \phi(x) \odot \phi(y)$ $F4$

5.2 Biblioteca para Processos

Todas as classes (operadores, axiomas, módulos, etc.) definidas na seção anterior deste capítulo podem ser usadas para especificar aplicações onde componentes de sistemas estão interligados, isto é, onde é necessário especificar concorrência e/ou comunicação entre componentes (processos).

Para implementar o conceito de reusabilidade na especificação de processos, agrupamos todas as classes em uma única biblioteca chamada ALG_PROCESS, como mostrado na Fig. 5.2. Nas duas próximas seções apresentamos exemplos que mostram a utilidade desta biblioteca.

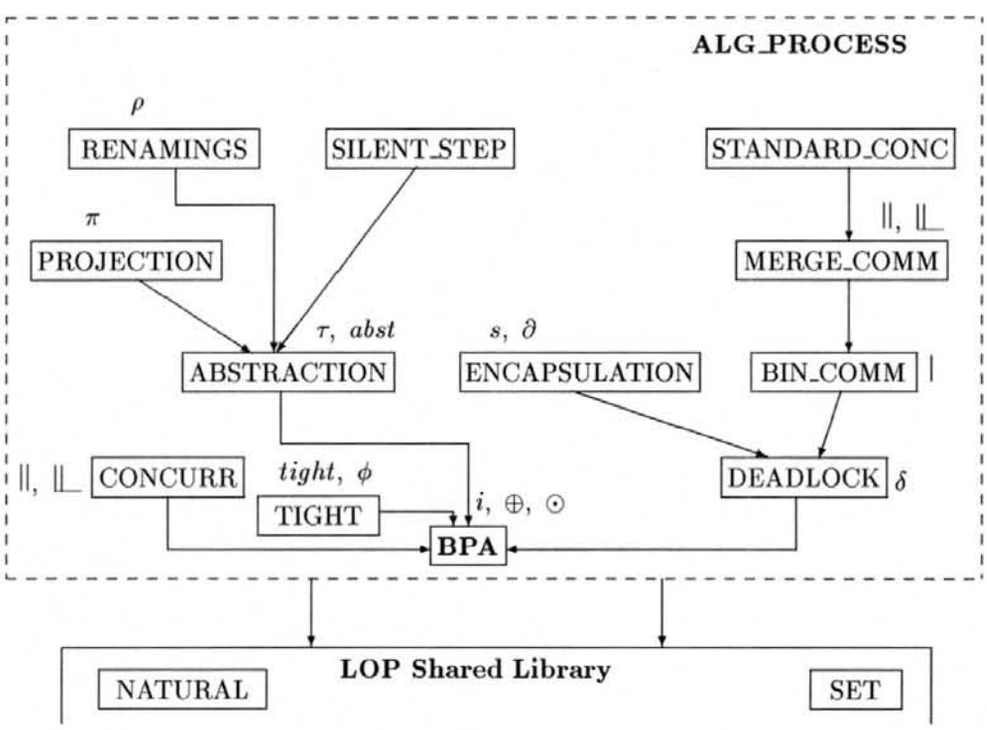


Figura 5.2 Biblioteca LOP para Processos

5.3 Um Exemplo Prático

Através deste exemplo mostraremos como usar a linguagem LOP para especificação de sistemas concorrentes incorporando módulos pré-definidos na biblioteca ALG_PROCESS. A seguir apresentamos os módulos principais da especificação do protocolo TRC (Transmitter/Receiver Communication Protocol) [3] usado para a transmissão de dados entre dois pontos P e C. Como mostrado na fig. 5.3, este protocolo consiste de quatro componentes: um transmissor (T), um canal de transmissão de dados (TRC), um receptor (R) e um canal de controle ou transmissão de conhecimento (RTC). A cada componente corresponde um processo. É óbvio que cada dado d oferecido por P deve ser recebido por C. Uma simples observação da fig. 5.3, verifica-se que os elementos básicos para a especificação são os processos, as portas, os dados a serem transmitidos e as mensagens. Para cada um deles será utilizado um sort: *Process, Ports, Data e Message.*

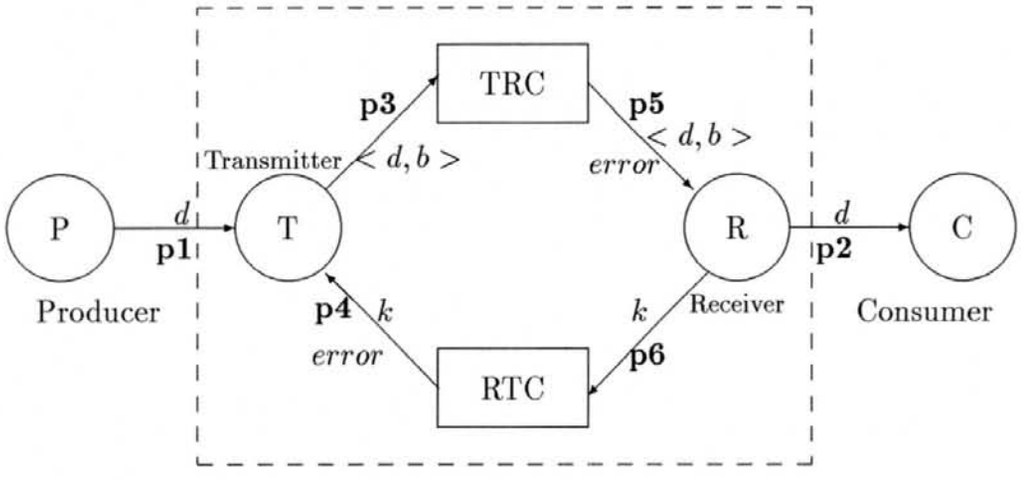


Figura 5.3 O Protocolo TRC

[3] O protocolo TRC é descrito em [TAN 81], sob o nome de Positive Acknowledgement with Retransmission - PAR

O primeiro módulo, apresenta a especificação de uma classe de dispositivos PORTS com três portas: *pa, pb* e *pc*, como visualizado na Figura 5.4. Observa-se na especificação que o sort *Ports* é gerado por três operadores: *pa*, *pb* e *pc*, isto é, o sort *Ports* é formado somente por três termos constantes: *pa*(), *pb*() e *pc*(). Doravante, e se não houver confusão entre operador e termo, a identificação dos termos que representam as portas também poderão ser feitas simplesmente escrevendo *pa*, *pb* e *pc*.

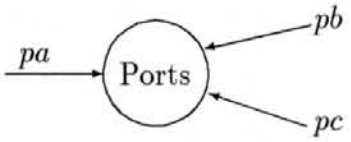


Figura 5.4 O dispositivo Ports

PORTS(pa, pb, pc) : **class**

introduces

pa : { } $\longrightarrow$ Ports

pb : { } $\longrightarrow$ Ports

pc : { } $\longrightarrow$ Ports

asserts

Ports **generated by** *pa, pb, pc*

Como mostrado na Figura 5.3, cada uma das seis portas fazem a conexão de dois processos e em cada porta uma interação entre os dois processos acontece. Três tipos de interação são definidos: leitura (*read*), envio de mensagens (*send*) e comunicação ($\|$, $|$). Os dois primeiros são especificados no módulo INTERACTION, e o terceiro é herdado do módulo MERGE_COMM (biblioteca ALG_PROCESS).

INTERACTION : **class**

introduces

read : Port, Message $\longrightarrow$ Process

send : Port, Message $\longrightarrow$ Process

O sort Message é o identificador do conjunto de mensagens que transitam pelos dois canais (TRC e RTC). As mensagens são de cinco tipos e quatro delas são constantes. A mensagem $frame(d, b)$ está formada pelo dado d a ser transmitido de P até C e por alguma informação booleana que indica a necessidade de retransmissão. As outras mensagens são: $ch\_error()$ (checksum-error, que indica uma mensagem danificada), $k()$ (mensagem de controle ou conhecimento), $time\_out()$ e $inter\_event()$ (para ações internas).

MESSAGES : **class**

introduces

frame	:	Data, Bool	⟶ Message
ch_error	:	{ }	⟶ Message
k	:	{ }	⟶ Message
time_out	:	{ }	⟶ Message
inter_event	:	{ }	⟶ Message

asserts

Message **generated by** *frame, ch_error, k, time_out, inter_event*

O **transmissor** T é especificado pelo módulo TRANSMITTER, da seguinte maneira: T é um host com três portas *pa*, *pb* e *pc*. Este lê um dado d da porta *pa* (na forma de mensagem: dado + informação booleana) [4] e depois é enviado pela porta *pb*. A seguir é executado o processo *wait_send*, quando então podem acontecer três coisas:

- *wait_send* recebe uma mensagem $k()$ pela porta *pc*, e então continua com a transmissão.
- *wait_send* recebe um checksum-error que indica que aconteceu uma falha na comunicação e que uma retransmissão deve seguir.

[4] Deve-se observar que, ainda que o elemento de interesse seja o dado d, este percorre os processos na forma de mensagens $d \equiv frame(d, b)$.

- Se nenhuma ação anterior acontece, um time-out ocorre e portanto uma retransmissão deve seguir.

TRANSMITTER(pa,pb,pc) : **class**

aggregation of BPA, INTERACTION, MESSAGES, PORTS

introduces

$$\begin{array}{rcll} transmitter & : & \{\,\} & \longrightarrow \text{Process} \\ read\_host & : & \text{Boolean} & \longrightarrow \text{Process} \\ send\_frame & : & \text{Data, Boolean} & \longrightarrow \text{Process} \\ wait\_send & : & \text{Message} & \longrightarrow \text{Process} \end{array}$$

asserts

for all b: Boolean, d: Data

$$read\_host(b) == read(pa, frame(d, true())) \odot send\_frame(d, b)$$

$$send\_frame(d, b) == send(pb, frame(d, b)) \odot wait\_send(frame(d, b))$$

$$wait\_send(frame(d, b)) == read(pc, k) \odot read\_host(\neg b) \oplus$$
$$(\ (read(pc, ch\_error()) \oplus time\_out()\ ) \odot send\_frame(d, b)\ )$$

$$transmitter() == read\_host(false())$$

O **canal de transmissão de dados** TRC (módulo DATATRC) é definido como o processo que espera pela entrada de uma mensagem na porta *pa*. A seguir deve ocorrer um dos seguintes subprocessos:

- A mensagem $< d, b >$ é enviada corretamente pela porta *pb*.
- A mensagem $< d, b >$ é defeituosa (mutilada) e portanto um checksum-error é comunicado pela porta *pb*.
- A mensagem $< d, b >$ é perdida por causa de um evento interno.

DATATRC(pa,pb,pc) : **class**

aggregation of BPA, INTERACTION, MESSAGES, PORTS

introduces

$$\begin{aligned} data\_transm &: \{\} \longrightarrow \text{Process} \\ frame\_transm &: \text{Message} \longrightarrow \text{Process} \end{aligned}$$

asserts

for all b:Boolean, d:Data

$$data\_transm() == read(pa, frame(d,b)) \odot frame\_transm(frame(d,b))$$

$$frame\_transm(frame(d,b)) == (\ send(pb, frame(d,b))\ \oplus\ send(pb, ch\_error())\ \oplus inter\_event()\ ) \odot data\_transm()$$

O processo **receptor** R é definido também como tendo três portas *pa*, *pb* e *pc*. Igual ao transmissor, R tem vários subprocessos. *wait_frame* espera pela chegada do dado d na forma de mensagem (frame) pela porta *pa*. Se um novo frame chegar, este é transmitido pela porta *pb* e logo depois é enviado a mensagem $k()$ pela porta *pc*. Se o canal RTC tiver problemas de funcionamento, a retransmissão do frame deve ocorrer e então uma mensagem $k()$ deve ser transmitida novamente. Se um checksum-error ocorre, então R espera até expirar o timer de T quando então, uma retransmissão deve acontecer. Este processo é especificado pelo módulo RECEIVER.

RECEIVER(pa,pb,pc) : **class**

aggregation of BPA, INTERACTION, MESSAGES, PORTS

introduces

$$\begin{aligned} receiver &: \{\} \longrightarrow \text{Process} \\ wait\_frame &: \text{Boolean} \longrightarrow \text{Process} \\ send\_ackn &: \text{Boolean} \longrightarrow \text{Process} \\ send\_host &: \text{Data, Boolean} \longrightarrow \text{Process} \end{aligned}$$

asserts

for all b:Boolean, d:Data

$$wait\_frame(b) == (read(pa, frame(d, \neg b)) \odot send\_ackn(b)) \oplus (read(pa, error()) \odot wait\_frame(b)) \oplus (read(pa, frame(d,b)) \odot send\_host(d,b))$$

$$send\_ackn(b) == send(pc, k) \odot wait\_frame(b)$$

$$send\_host(d,b) == send(pb, frame(d, true())) \odot$$
$$send(pc, k()) \odot wait\_frame(\neg b)$$
$$receiver() == wait\_frame(false())$$

O canal de controle RTC lê pela porta *pa* a mensagem $k()$ e logo acontece uma das seguintes possibilidades:

- $k()$ é enviada corretamente pela porta *pb*, ou
- Se $k()$ está defeituosa, então um checksum-error é enviado pela porta *pb*, ou
- $k()$ é perdida pela ocorrência de um evento interno a RTC, neste caso, todo o processo RTC ocorre novamente.

ACKNTRC(pa,pb,pc) : **class**

aggregation of BPA, INTERACTION, MESSAGES, PORTS

introduces

$ackn\_transm$: { } $\longrightarrow$ Process

asserts

$$ackn\_transm() == read(pa, k()) \oplus (send(pb, k())$$
$$\oplus send(pb, ch\_error()) \oplus inter\_event()) \odot ackn\_transm()$$

Até aqui foram especificados quatro processos independentemente uns dos outros, exceto pelos nomes usados. Por exemplo, o transmissor e o receptor utilizam a mesma porta *pa* para a entrada da mensagem $< d, b >$ e a mesma porta *pb* para a saída. Agora, usando o conceito de *especialização* da linguagem LOP, podemos especificar separadamente os componentes do protocolo TRC integrando-os em função de suas seis portas $p_1, \ldots, p_6$.

T_TRC : **class**

specialization of TRANSMITTER(p1 **for** pa, p3 **for** pb, p4 **for** pc)

TRC_TRC : **class**

specialization of DATATRC(p3 **for** pa, p4 **for** pb)

R_TRC : **class**

specialization of RECEIVER(p5 **for** pa, p2 **for** pb, p6 **for** pc)

RTC_TRC : **class**

specialization of ACKNTRC(p6 **for** pa, p4 **for** pb)

Seja $s1 = \{send(p_j, frame(d,b))|read(p_j, frame(d,b))/j = 3,4,5,6 \;\; b \in Bool\}$ o conjunto de processos com tentativas de comunicação sem sucesso, e seja também

$$s2 = \{send(p_j, frame(d,b))|read(p_j, frame(d,b))/j = 3,4,5,6 \;\; b \in Bool\} \cup \text{IP}$$

onde IP $= \{time\_out(), inter\_event()\}$, o conjunto de processos que acontecem nas portas p3, p4, p5 e p6. Então o protocolo TRC pode ser definido encapsulando o conjunto s_1 e abstraindo depois o conjunto s_2, pois o único comportamento externo que é de interesse, é aquele que acontece nas portas p1 e p2.

Assim, finalmente apresentamos a especificação final do protocolo TRC.

TRC_protocol : **class**

aggregation of T_TRC, TRC_TRC, R_TRC, RTC_TRC, ACP, ABSTRACTION

introduces

$trc\_protocol$: { } $\longrightarrow$ Process

asserts

$trc\_protocol() == abst(s_2, \partial(\; s_1, transmiter() \| trc() \| receiver() \| rtc()) \;)$

implies

ENCAPSULATION(s **for** s_1), ABSTRACTION(s **for** s_2)

converts *abst*

5.4 Sistemas de Axiomas Especiais

A seguir apresenta-se alguns sistemas de axiomas avançados que podem ser chamados de "especiais" pela sua importância na especificação de certas aplicações.

- **Fusão Livre de Processos - PA**

  PA é um dos sistemas mais simples para descrever a fusão de processos sem comunicação, chamado também de *Fusão Livre de Processos.* Este módulo é formado pelos axiomas A1-A5 e M1-M4 da seção 5.1.

  PA : **class**

  aggregation of BPA, CONCURR

- **Álgebra de Processos Comunicantes - ACP**

  O sistema ACP é considerado um sistema de *uso geral* e é derivado do CCS de Milner (ver [BER 84] e [BER 85]). Este é definido para processos que podem se comunicar, pela partilha (execução simultânea) de ações. ACP é uma especificação algébrica de cooperação de processos assíncronos através de comunicação síncrona.

  O módulo ACP é uma extensão conservativa de BPA, pela inclusão dos operadores δ (que indica a falha da comunicação entre duas ações), $\|$ e $\lfloor\!\lfloor$ para concorrência, $|$ para comunicação e ∂_s para encapsulamento de um processo. O sistema ACP está formado pelos axiomas A1-A7, C1-C3, CM1-CM9 e D1-D4 da seção 5.1.

  ACP : **class**

  aggregation of MERGE_COMM, ENCAPSULATION(s)

- **Álgebra de Processos Comunicantes com Abstração - ACP_τ**

  Este sistema é definido em [BER 85] e [BER 89] e é uma extensão conservativa do sistema ACP. Processos em ACP_τ incluem as τ-ações. ACP_τ consiste de ACP junto com as bem conhecidas *Leis τ de Milner* T1-T3. A diferença de ACP_τ para CCS, esta no fato que, ACP_τ especifica o comportamento das τ-ações em relação com o operador de comunicação $|$, um operador que não esta presente em CCS. Outro aspecto importante de ACP_τ (em relação a CCS) é a separação entre abstração e comunicação: abstração é executada pelo operador $abst_s$. ACP_τ compreende ACP mais os axiomas T1-T3, TM1, TM2, TC1-TC4, TS1-TS5 e DT da seção 5.1.

  ACP_τ : **class**

  aggregation of ACP, SIL_MER,

  asserts

  $\partial(s, i_\tau) == i_\tau$ *DT*

- **Cooperação Síncrona de Processos - ASP**

  O sistema de axiomas ASP descreve cooperação síncrona de Processos no sentido de como a cooperação dos processos $x_1, \ldots, x_n$, denotado por $x_1|x_2|\cdots|x_n$ é definido em [BER 84].

  Pode-se dizer que ASP tem sua origem no sistema ACP pela exclusão do axioma CM1 de ACP:

  $$x \| y == (x \lfloor\!\lfloor y) \oplus (y \lfloor\!\lfloor x) \oplus (x|y)$$

  O módulo MERGE_COMM é substituido pelos axiomas SM1-SM5. O sistema ASP lembra o SCCS de Milner [MIL 83].

  ASP : **class**

  aggregation of BIN_COMM

  asserts

  for all a, b: Atoms, x, y: Process

$$(x \oplus y)|z == (x|z) \oplus (y|z) \qquad SM1$$
$$x|(y \oplus z) == (x|y) \oplus (x|z) \qquad SM2$$
$$(i_a \odot x)|(i_b \odot y) == (i_a|i_b) \odot (x|y) \qquad SM3$$
$$i_a|(i_b \odot y) == (i_a|i_b) \odot y \qquad SM4$$
$$(i_a \odot x)|i_b) == (i_a|i_b) \odot x \qquad SM5$$

- **Merge com Exclusão Mútua de Regiões Compactas - AMP**

AMP é uma extensão conservativa do módulo CONCURR e é definido como um sistema de axiomas para processos com regiões compactas sem comunicação.

AMP : **class**

aggregation of CONCURR, TIGHT

asserts

for all x, y, z: Process

$$tight(x)\|\!\llcorner y == tight(x) \odot y \qquad TRM1$$
$$(tight(x) \odot y)\|\!\llcorner z == tight(x) \odot (x\|y) \qquad TRM2$$

- **AMP(:)**

Este sistema também é definido em [BER 84] com o propósito de estender o significado do operador $\odot$ em regiões compactas. Para isto é definido o operador " : " (produto 'tight'). O termo $x : y$ é interpretado em forma similar a $x \odot y$, porém, com a diferença que, em uma composição paralela nenhuma ação de um processo paralelo pode ser entrelaçado entre os processos x e y. Assim, em $x : (y \odot z \ \oplus w)$ depois de x e antes de $y \odot z \ \oplus w$ nenhuma interrupção é possível.

AMP(:) : **class**

aggregation of TIGHT, CONCURR

introduces

$\_:\_:$ Process $\longrightarrow$ Process

asserts

for all x, y, z: Process

$tight(x \odot y) == tight(x) \odot tight(y)$ $TR4$

$tight(x : y) == tight(x) : tight(y)$ $TR5$

$(x \oplus y) : z == (x : z) \oplus (y : z)$ $AT1$

$(x : y) : z == x : (y : z)$ $AT2$

$(x : y) \odot z == x : (y \odot z)$ $AT3$

$(x \odot y) : z == x \odot (y : z)$ $AT4$

$(i_a : x) \between y == i_a : (x \between y)$ TRM

$\phi(x : y) == \phi(x) \odot \phi(y)$ $F5$

- **ACMP**

O sistema ACMP é uma combinação dos módulos ACP e AMP(:) para processos com regiões compactas e *com comunicação.* Este módulo também é definido em [BER 84].

ACMP : **class**

aggregation of TIGHT, ENCAPSULATION, MERGE_COMM

asserts

for all a, b: Atoms, x, y, z: Process

$(x \oplus y) : z == (x : z) \oplus (y : z)$ $AT1$

$(x : y) : z == x : (y : z)$ $AT2$

$(x : y) \odot z == x : (y \odot z)$ $AT3$

$(x \odot y) : z == x \odot (y : z)$ $AT4$

$\delta() : x == \delta()$ $AT5$

$(i_a : x) \between y == i_a : (x \between y \oplus x|y)$ $CTRM1$

$(i_a : x)|(i_b : y) == (i_a|i_b) : (x|y)$ $CTRM2$

$(i_a : x)|(i_b \odot y) == (i_a|i_b) : (x \between y \oplus x|y)$ $CTRM3$

$$(i_a : x)|i_b == (i_a|i_b) : x \qquad CTRM4$$

$$tight(x \odot y) == tight(x) \odot tight(y) \qquad TR4$$

$$tight(x : y) == tight(x) : tight(y) \qquad TR5$$

$$\phi(x : y) == \phi(x) \odot \phi(y) \qquad F5$$

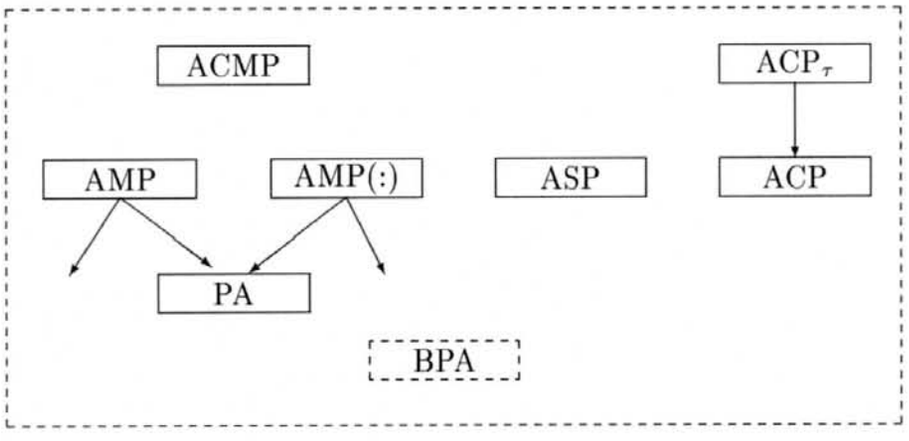


Figura 5.5 Sistemas de Axiomas Especiais

6 MÉTODO DE ESPECIFICAÇÃO FORMAL

A descrição formal (especificação) de um Sistema de Software torna-se útil e efetiva, após a realização de um processo que compreende dois aspectos importantes: *como fazer* esta descrição (método) e *como expressar* esta descrição (linguagem) aos usuários da mesma. No que diz respeito ao segundo aspecto, nos capítulos anteriores já foram definidos os componentes principais através da linguagem LOP. Em relação ao método, neste capítulo apresentaremos alguns conceitos da nossa proposta.

6.1 Antecedentes

Até hoje, dois tipos de metodologias para especificação de sistemas de software tem sido propostas e usadas. O primeiro tipo compreende metodologias associadas diretamente a construção de especificações *formais* usando linguagens de especificação formal. O segundo tipo, é um conjunto de metodologias (informais e semi- formais) para Análise e Projeto de sistemas orientados a objetos.

Nos últimos anos duas abordagens foram usadas para a construção de especificações formais: os métodos orientados a modelos (VDM, RAISE, CSP, etc.) através do desenvolvimento "top-down", e os métodos orientados a propriedades, através da abordagem "bottom-up".

VDM e RAISE usam *desenvolvimento de especificações por etapas.* VDM incorpora *Desenvolvimento por Reificação de Dados* (agregando detalhes em cada etapa) e *Desenvolvimento por decomposição de Operações* (criando obrigações de prova).

RSL (RAISE Specification Language) usa *refinamento por etapas* e uma relação de refinamento chamada *relação de implementação estática.* Em cada etapa

existe refinamento para esquemas e objetos, para variáveis e canais, para tipos, para valores e para axiomas.

Os métodos orientados a propriedades como OBJ, Larch, ASL, Axis, etc. por terem a característica de ser modulares e usar o mecanismo de parametrização, quase sempre possuem bibliotecas de especificações básicas. Isto permite e facilita a construção "bottom-up" de outros módulos baseados nos já existentes, supostamente refinados e bem definidos. Larch usa o método de *construção incremental* de especificações.

Por outro lado, ainda que os conceitos de orientação a objetos apareceram antes da década dos 80 e relacionados a programação, somente nestes últimos anos surgiram as metodologias (não formais) dentro da classificação de desenvolvimento de software clássico de Análise, Projeto e Implementação Orientados a Objetos e hoje ainda não atingiram consenso e maturidade.

Em [HEN 90], três metodologias de desenvolvimento de sistemas são analizadas:

- **O-O-O** (Object-Oriented analysis, Object-Oriented design, Object-Oriented implementation).
- **F-O-O** (Functional analysis, Object-Oriented design, Object-Oriented implementation).
- **O-O-F** (Object-Oriented analysis, Object-Oriented design, procedural implementation).

Também [FIC 92] faz um estudo comparativo e crítico de algumas metodologias para análise e projeto de sistemas (convencionais e orientadas a objetos):

- Especificação de Requisitos orientada a objetos, [BAI 89]:
- Análise Orientada a Objetos, [COA 92]:

- Análise Orientada a Objetos, [SHL 88], [SHL 92]:
- Projeto Estruturado Orientado a Objetos, [WAS 89]
- Projeto Orientado a Objetos, [BOO 89]
- Projeto Dirigido por Responsabilidades RDD, [WIR 90], [WIR 90a]

Estas duas últimas referências mostram a grande preocupação por definir metodologias ideais para serem usadas na construção de software orientado a objetos. Devemos salientar também que, as metodologias acima mencionadas, foram estabelecidas pensando somente em um processo de solução com características "orientadas a objetos" sem considerar aspectos de natureza formal.

6.2 O Método

Um dos objetivos desta tese é estabelecer uma metodologia (seqüência de tarefas) que permita, a partir de um problema real, construir especificações usando a linguagem LOP. A idéia é usar os conceitos existentes sobre análise e projeto orientados a objetos e definir um conjunto de passos que deve-se seguir até obter-se uma especificação formal do sistema a ser construído e expresso em módulos LOP.

O método é de natureza evolutiva semiformal, e está baseado nas abordagens apresentadas em [BAI 89] e [McG 92] e, pode ser resumido nos seguintes passos:

- **PASSO 1**

  Identificar as entidades chaves do domínio do problema.

  A partir dos requisitos do sistema, devem ser identificadas as principais identidades presentes no problema, isto é, identificar os conceitos

básicos que caracterizam o domínio. Estas entidades podem ser objetos reais (disquete, livro, automóvel, etc.) ou conceitos abstratos (processos, funções, incidentes (ocorrência de atividades), interações, etc.).

- **PASSO 2**

  Identificar os relacionamentos entre entidades

  Entidades identificadas na etapa anterior não são items isolados, portanto, deve-se identificar os relacionamentos (interações) entre elas. Duas entidades podem ter mais de um relacionamento entre si. Uma entidade poderá ter relacionamentos com mais de uma entidade. A identificação dos principais relacionamentos entre entidades é muito importante para definir os diferentes mecanismos de herança presentes no problema. Uma ferramenta útil nesta etapa que ajuda a visualizar os diferentes relacionamentos, são os diagramas Entidade/Relacionamento (E-R).

- **PASSO 3**

  Identificar os atributos de cada entidade

  Visto que, atributos indicam o comportamento e o estado de uma entidade, estes serão de natureza descritiva:

  NomeEntidade

  NomeAtributo1 - descrição

  NomeAtributo2 - descrição

  . . . . . .

  A identificação destos atributos são muito importantes para a definição das classes da especificação.

- **PASSO 4**

  Identificar especializações, agregações e composições

  É importante identificar um segundo tipo de relacionamento entre entidades: a relação de componente. Os atributos identificados na etapa

anterior facilitam a representação de abstrações a partir de elementos (entidades) pequenos com atributos comuns. Desta forma abstrações podem ser construídas em forma incremental. Especializações representam abstrações em uma camada do mesmo nível. Agregações e composições representam abstrações em um nível superior ao de seus componentes.

- **PASSO 5**

  Identificar as entidades chaves do sistema a ser desenvolvido

  Nesta etapa, deve-se identificar as entidades chaves a serem usadas na solução do problema, isto é, na construção de um sistema baseado-em-computador. Entidades identificadas no passo 1, bem como entidades relacionadas com processamento de dados, interface do usuário, etc. devem ser consideradas. Estas entidades serão transformadas em "sorts" para definir as diferentes signaturas nas especificações LOP.

- **PASSO 6**

  Identificar as classes do sistema

  Classes representam uma única idéia ou conceito bem definido e fornecem uma unidade natural para reusabilidade. Nesta etapa, deve-se identificar as classes a serem construídas bem como as classes existentes em uma biblioteca LOP. [JOH 88] apresenta um conjunto de dicas que devem ser consideradas na identificação-construção de classes.

- **PASSO 7**

  Especificar as classes usando LOP

  Usando o método de construção incremental e a linguagem LOP definida nos capítulos anteriores especificar as classes identificadas na etapa anterior.

No método acima proposto, obviamente, não existe regra fixa para o processo de identificação (de entidades, classes, atributos, relacionamentos, etc.), porém, a experiência prática ajuda muito.

6.3 Aplicação: O Trânsito em uma interseção

Para ilustrar o método, consideremos o exemplo proposto em [McG 92], sobre o desenvolvimento de um sistema de software que deverá controlar o trânsito em uma interseção, como mostrado na Figura 6.1, através de luzes coloridas (semáforos) baseados em impulsos emitidos a partir de sensores.

Por se tratar somente de um exemplo usado para fins didáticos e não para implementação, em cada etapa do método mostraremos em forma parcial as diferentes tarefas a serem realizadas.

Os requisitos para o sistema de software é descrito a seguir.

- **Requisitos**

  O software rodando em uma caixa de controle (controlador ou caixa controladora) baseada em um microcomputador PC, controlará o trânsito em uma interseção através de luzes de trânsito (sinaleiras) baseado nas entradas de impulsos emitidos por sensores colocados em cada uma das rotas de trânsito na interseção. Quando um veiculo passar (ou parar) sobre um sensor, este modificará um bit em um determinado endereço da memória da caixa de controle. Este endereço (byte address) é único para cada sensor. Sensores eletrônicos ou mecânicos podem ser usados, porém, ambos devem se comunicar com a caixa controladora da mesma maneira. A controladora pode recompor o bit para um determinado sensor e logo detetar a presença de um veículo escolhendo o correspondente bit até este ser atualizado. Se um bit é marcado e um

veículo está detido ou movimentando-se sobre um sensor, então o bit de novo é marcado. A caixa de controle também contém um relogio que pode ser lido para determinar a data, a hora no dia e um marcador de tempo que pode ser marcado até 1024 segundos. Quando o marcador de tempo chegar a 0 (contagem regressiva), este gera uma interrupção no microprocessador.

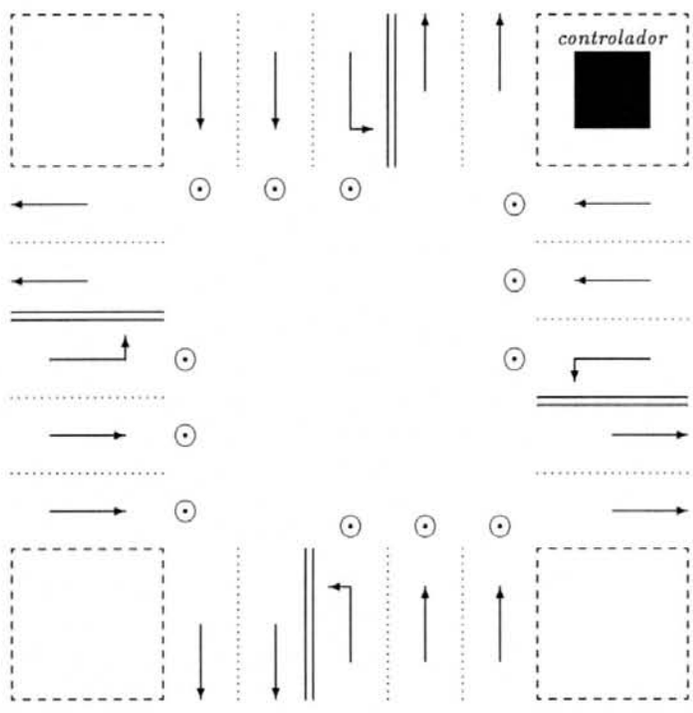


Figura 6.1 Trânsito em uma interseção

A interseção contém dois tipos de sinais de trânsito. Um tipo é o sinal padrão com luzes vermelho, amarelo e verde, e é usado para controlar o trânsito em faixas retas o "voltar a direita".. O outro tipo de sinal é o padrão mais uma cor adicional, uma seta verde sinalando à esquerda e é usado em cada uma das faixas "voltar a esquerda" para indicar uma volta a esquerda protegida, isto é, uma seta verde indica que veículos nesta faixa podem voltar a esquerda sem produzir trânsito imediato, e

uma circular verde indicando que voltar a esquerda é pemitido e trânsito imediato só na faixa da direita. As luzes são controladas via saídas mapeadas desde a memória. A cada sinal corresponde um byte na memória do controlador, e um bit fixo em cada byte controla uma luz em uma sinaleira. Marcando um bit produz ON na luz associada e, atualizando este bit produz OFF.

A responsabilidade da caixa controladora é sequenciar os sinais de tal forma que o fluxo de trânsito é mantido na interseção em uma maneira segura e uniforme. Para trânsito em direções opostas (norte/sul e leste/oeste), se um veículo está em uma faixa "voltar a esquerda", então o controlador deverá fornecer uma seta "voltar a esquerda" por TL segundos ou até o sensor desta faixa indicar nenhum veículo mais, quando o sinal deverá mostrar um verde circular até o trânsito na faixa reta adjacente mostrar luz vermelha. Quando uma luz verde de voltar a esquerda terminar, as sinaleiras para o trânsito na direção oposta imediata deverão indicar verde circular durante TS segundos. Este tempo deverá ser estendido se a sinaleira na direção oposta ainda não permaneceu verde por TS segundos. Nota: Se nenhum trânsito é detetado pelos sensores durante TX segundos, então todas as sinaleiras deverão ser mostradas com luz amarelo e logo depois com luz vermelha se foi detetado pelos sensores trânsito nas direções transversais.

Se nenhum trânsito é detetado em qualquer faixa, então as luzes deverão ser seqüenciadas cada TN segundos e as sinaleiras "voltar a esquerda" deverão mostrar luz vermelha em todas as direções.

A caixa controladora suporta três tipos de interrupções:

- *Uma interrupção* **power on/start** *usada para inicializar a caixa controladora (trânsito nas direções norte/sul)*

- *Uma interrupção* **test** *usada para verificar todas as sinaleiras através de uma seqüência rápida de todos os possíveis estados. Assume-se que todos os sensores tem sido apanhados em erro e que TS e TL tem valores de 10 e 5 segundos respectivamente.*
- *Uma interrupção de* **espera**, *usada para controlar a interseção quando novos programas estão sendo carregados no controlador.*

O volume de trânsito nesta interseção é projetado a se duplicar nos próximos dois anos. O controle de trânsito poderá precisar de algumas modificações para incorporar tempo e volume de veículos nos algoritmos de controle. Estas considerações deverão ser determinadas na base das medidas feitas periodicamente nos próximos 18 meses.

PASSO 1

Identificar as entidades chaves do domínio do problema

controlador

sinaleiras

sensores

rotas ou faixas de trânsito

interseção

veículo

memória do controlador

relógio = data + tempo

direção = norte, sul, leste, oeste

PASSO 2

Identificar os relacionamentos entre entidades

A Figura 6.2 mostra um diagrama E-R identificando os relacionamentos controlador-relógio, controlador-sensor, sensor-faixa, faixa-sinaleira, faixa-direção, direção-interseção

e interseção-controlador. Para uma implementação completa do sistema, outros relacionamentos também devem ser identificados, usando outros diagramas E-R.

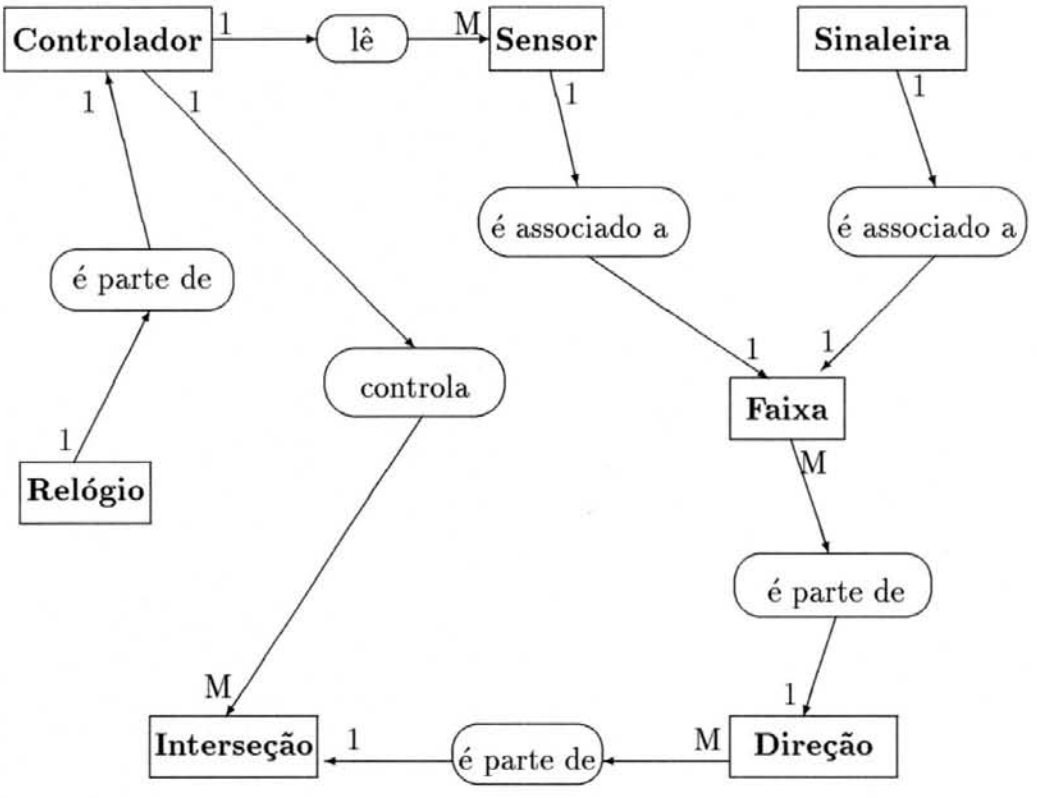


Figura 6.2 Um diagrama E-R para o problema do trânsito numa interseção

PASSO 3

Identificar os atributos de cada entidade

Controlador

estado inicial -

relogio -

memoria -

Sinaleira

identificador - número inteiro entre 1 e 12

tipo - padrão, "volta a esquerda"

estado -

próximo estado -

estado default -

time-step -

Sensor

identificador - número inteiro entre 1 e 12

faixa - reta, "volta a esquerda"

tipo - eletromagnetico, pressão

sinal - 0, 1

faixa (rota)

identificador - número inteiro entre 1 e 12

tipo - reto, "volta a esquerda"

estado - livre, ocupada

Memória

estado inicial -

estado atual -

Relógio

data - dia, mes, ano

tempo - hora, minuto, segundo

Direção

identificador - norte, sul, leste, oeste

proxima -

anterior -

Falta identificar os atributos de veículo e interseção. Atributos adicionais para as entidades acima também podem ser identificados.

PASSO 4

Identificar especializações, agregações e composições

a). agregações:

controlador = switchs + clock + memória

. . . . . .

b). composições:

direção = Norte + Sul + Leste + Oeste)

. . . . . .

c). especializações:

sinaleira (padrão)

sinaleira (volta a esquerda)

faixa (reta)

faixa (volta a esquerda)

. . . . . .

Ver Figura 6.3

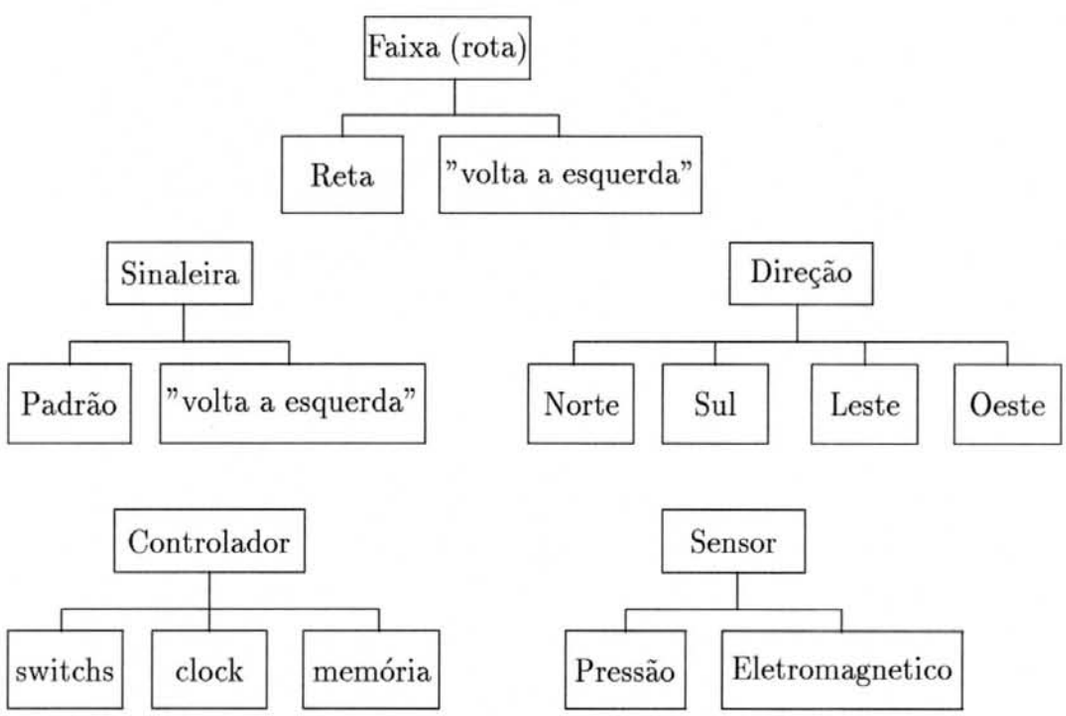


Figura 6.3 Especializações, Composições, ...

PASSO 5

Identificar as entidades chaves do sistema a ser desenvolvido

Processos

controlador

rota = reta + "volta a esquerda"

identificador de rota

interseção

direção (orientação

sensor

sinaleira = normal + "volta a esquerda"

luzes de trânsito

interrupção

tempo

memória

bit

Byte

Observe-se que, além das entidades do passo 1, nesta etapa foram identificadas outras identidades (Processos, tempo, bit, etc.).

PASSO 6

Identificar as classes do sistema

controlador

sinaleira

relógio

direções

sensor

memória

luz de trânsito

PASSO 7

Especificar as classes usando LOP (ver Figura 6.4)

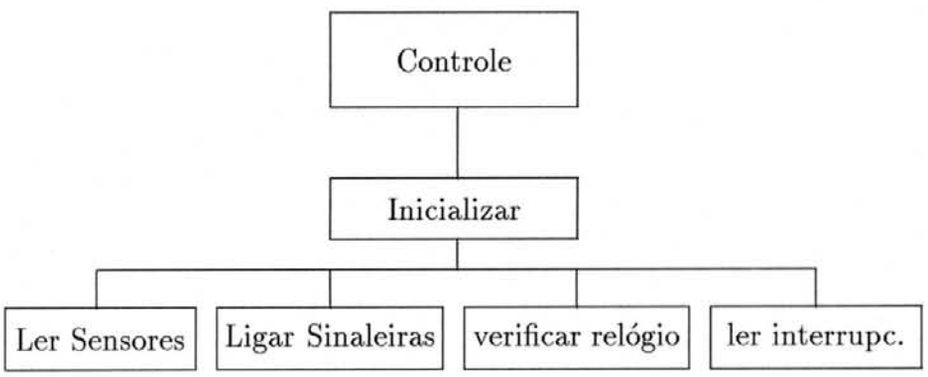


Figura 6.4 O processo de Controle das Sinaleiras na interseção

INTERSECTION : **class**

aggregation of DIRECTIONS

TrafficIntersection **enumeration of** lane1, ..., lane12

LANE: **class**

introduces

$lright, lturn$: $\{\}$ $\longrightarrow$ Lane

$lfree, lbussy$: LaneId $\longrightarrow$ Bool

$laneType$: LaneId $\longrightarrow$ Lane

$laneState$: LaneId $\longrightarrow$ FreeBussyLane

asserts

Lane **generated by** *lright, ltunr*

CLOCK : **class**

introduces

$initClock$: $\{\}$ $\longrightarrow$ Process

$checkClock$: $\{\}$ $\longrightarrow$ Process

TL : $\{\}$ $\longrightarrow$ Time

TS : $\{\}$ $\longrightarrow$ Time

TX : $\{\}$ $\longrightarrow$ Time

TN : $\{\}$ $\longrightarrow$ Time

asserts

.....

MEMORY : **class**

introduces

$setMemState$: $\{\}$ $\longrightarrow$ Array

$setBitMem$: LaneId, Bool $\longrightarrow$ Process

$readInterrup$: $\{\}$ $\longrightarrow$ Process

asserts

for all id: LaneId, b: Bool

$setBitMem(id, b) ==$ .....

.....

$readInterrup() ==$ .....

SENSOR : **class**

introduces

$readSensor$: LaneId $\longrightarrow$ Bool

asserts

$readSensor() ==$ .....

RIGHT_TRAFFIC_LIGHT : **class**

introduces

$$\begin{aligned} green &: \{\} &&\longrightarrow \text{RightColor} \\ red &: \{\} &&\longrightarrow \text{RightColor} \\ amber &: \{\} &&\longrightarrow \text{RightColor} \\ nextrc &: \text{RightColor} &&\longrightarrow \text{RightColor} \end{aligned}$$

asserts

RightColor **generated by** *green, red, amber*

$nextrc(green()) == amber()$

$nextrc(red()) == green()$

$nextrc(amber()) == red()$

TURN_TRAFFIC_LIGHT : class

introduces

$$\begin{aligned} green &: \{\} &&\longrightarrow \text{TurnColor} \\ greenArrow &: \{\} &&\longrightarrow \text{TurnColor} \\ red &: \{\} &&\longrightarrow \text{TurnColor} \\ amber &: \{\} &&\longrightarrow \text{TurnColor} \\ nextrc &: \text{TurnColor} &&\longrightarrow \text{TurnColor} \end{aligned}$$

asserts

TurnColor **generated by** *green, greenArrow, red, amber*

$nextrc(green()) == amber()$

$nextrc(red()) == greenArrow()$

$nextrc(amber()) == red()$

$nextrc(greenArrow()) = green()$

TRAFFIC_LIGHT : class

aggregation of LANE, MERGE_COMM

$$
\begin{array}{rcll}
setFlashLight & : & \{\,\} & \longrightarrow \text{Process} \\
setFlash1 & : & \text{LaneId} & \longrightarrow \text{Process} \\
setFlash2 & : & \text{LaneId} & \longrightarrow \text{Process} \\
lightGreen & : & \text{LaneId} & \longrightarrow \text{Process} \\
setGreenTurn & : & \text{LaneId} & \longrightarrow \text{Process} \\
setGreenRight & : & \text{LaneId} & \longrightarrow \text{Process} \\
setColor & : & \text{LaneId, TrafficColor} & \longrightarrow \text{Process}
\end{array}
$$

asserts

for all id : LaneId

$$setFlashLight() == setFlash1(1) \parallel setFlash1(2) \parallel setFlash1(3) \parallel$$
$$setFlash2(4) \parallel setFlash2(5) \parallel setFlash2(6) \parallel setFlash1(7) \parallel$$
$$setFlash1(8) \parallel setFlash1(9) \parallel setFlash2(10) \parallel setFlash2(11) \parallel$$
$$setFlash2(12)$$

$$setFlash1(id) == lightGreen(id) \odot lightAmber(id) \odot lightRed(id) \odot setFlash1(id)$$

$$setFlsh2(id) == lightRed(id) \odot lightGreen(id) \odot lightAmber(id) \odot setFlash2(id)$$

$$lightGreen(id) == \textbf{if } laneType(id) = lturn() \textbf{ then } setGreenTurn(id) \textbf{ else } setGreenRight(id)$$

$$setGreenTurn(id) == setColor(id, greenArrow()) \odot ((waitTL() \odot setColor(id, green()) \odot waitTS()) \oplus (NoVehicle() \odot setColor(id, green()) \odot waitAdjacentRed()))$$

$$setGreenRight(id) == setColor(id, green()) \odot (waitTS() \oplus waitTX())$$

CONTROLLER : class

aggregation of TRAFFIC_LIGHT, MEMORY, CLOCK

introduces

$$controller \ : \ \{\} \longrightarrow \text{Process}$$

$$initController \ : \ \{\} \longrightarrow \text{Process}$$

$$read12sensors \ : \ \{\} \longrightarrow \text{Process}$$

asserts

$$controller() == initController() \odot (\ read12sensors() \parallel setFlashLight() \parallel checkClock() \parallel readInterrup() \ )$$

$$initController() == (\ setColor(1, green()) \parallel setColor(2, green()) \parallel setColor(3, greenArrow()) \parallel setColor(4, red()) \parallel setColor(5, red()) \parallel setColor(6, red()) \parallel setColor(7, green()) \parallel setColor(8, green()) \parallel setColor(9, greenArrow()) \parallel setColor(10, red()) \parallel setColor(11, red()) \parallel setColor(12, red()) \ ) \parallel initClock()$$

$$read12sensors() == (setBitMem(1, readSensor(1)) \parallel setBitMem(2, readSensor(2)) \parallel setBitMem(3, readSensor(3)) \parallel setBitMem(4, readSensor(4)) \parallel setBitMem(5, readSensor(5)) \parallel setBitMem(6, readSensor(6)) \parallel setBitMem(7, readSensor(7)) \parallel setBitMem(8, readSensor(8)) \parallel setBitMem(9, readSensor(9)) \parallel setBitMem(10, readSensor(10)) \parallel setBitMem(11, readSensor(11)) \parallel setBitMem(12, readSensor(12)) \ ) \odot read12sensors()$$

7 CONCLUSÕES

Como resultado do desenvolvimento da tese consideraremos três aspectos importantes: a definição de uma nova linguagem de especificação formal, as diferenças com outras linguagens existentes e, a definição de um método de desenvolvimento de software que finalize com uma especificação na linguagem LOP.

A. **A Linguagem**. Foi definida a linguagem de especificação LOP, integrando três conceitos modernos de Engenharia de Software: especificação algébrica, orientação a objetos e especificação de processos (concorrência).

   Entre as principais características da linguagem LOP, podemos enumerar as seguintes:

   (a) Tem o formato da maioria das linguagens algébricas, isto é, cada especificação LOP tem duas partes: signatura e axiomas (equações).

   (b) Especificações LOP são independentes de qualquer processo de execução. Isto significa que, na forma como foi definida a linguagem, não será possível fazer testes com módulos LOP.

   (c) Cada especificação LOP tem a estrutura modular e a maioria delas são construídas em forma incremental, pela adição de novas propriedades (operadores e/ou axiomas) a outras especificações existentes.

   (d) Foram incluídas algumas construções sintáticas, herdadas da linguagem Larch, para fins de verificação sintática e para determinar as caraterísticas dos operadores da signatura (**generated by** e **partitioned by**). Verificação sintática redundante é incorporada através de cláusulas no final de cada especificação, permitindo tratamento indireto de erros e exceções.

(e) A importação/exportação de módulos é realizada usando conceitos de orientação a objetos. Cada módulo LOP é uma classe. Cada classe tem objetos sintáticos e objetos semânticos. A herança entre classes é definida somente entre classes construídas em forma incremental: umas a partir de outras. Para isto é usado o conceito matemático chamado de *mecanismo de modificação incremental* B = A + m : a entidade A é transformada na entidade B usando o modificador m.

(f) Os elementos de um sistema concorrente: os processos e os agentes, são considerados em um nível muito abstrato, de tal forma que sua semântica é muito simples. Isto permite que, usando a linguagem LOP, seja possível especificar em forma homogênea, tanto sistemas sequenciais bem como sistemas concorrentes.

(g) Foram apresentados muitos exemplos e duas aplicações (incompletas) pensando principalmente em aspectos didáticos.

B. **Outras Linguagens**

Todas as linguagens de especificação formal são classificadas em duas categorias: orientadas a modelos e orientadas a propriedades. Qualquer comparação efetiva entre linguagens deve ser entre aquelas que pertencem à mesma categoria. Linguagens como Z, Object-Z, MooZ, VDM, O=M, OOZE são orientadas a modelos. Entre as linguagens (ou propostas de linguagens) orientadas a propriedades que podemos considerar para efeitos de comparação temos: OBJ3 [KIR 88], LSL [GUT 90], RSL [GEO 90], Maude [MES 90], SMoLCS-SCDS [AST 91], ASF [BER 89a], ASL [van 89a], ABEL [DAH 86], ACT ONE [EHR 85] e LOTOS [BOL 87]. Um trabalho comparativo entre algumas delas já foi realizado em [CAS 92].

- OBJ3 e LSL são algébricas, porém a cada módulo principal está associado outros módulos: teorias e visões em OBJ3, módulo interface em Larch. ASL tem quatro tipos de módulos: module, schema, instantiate e run. Módulos LOP são simples e não tem associado nenhuma outra especificação. OBJ3, LSL e ASL não tem OO nem incorporam especificação de processos.
- RSL integra espec. algébrica, orientação a modelos, definições axiomáticas, usa estilos aplicativos, imperativos, concorrente e modular e tem muitas ferramentas automatizadas. O aspecto misturado de especificações RSL em muitos estilos descaracteriza o aspecto formal da linguagem. ABEL tem quase o mesmo estilo (imperativo).
- Maude é uma extensão de OBJ3 e incorpora uma lógica de reescrita concorrente que permite executar especificações.
- SMoLCS-SCDS usa tipos de dados abstratos dinâmicos junto com álgebras dinâmicas baseados em sistemas de transição rotulados. Incorpora todo o CCS para tratamento de concorrência.
- ASF usa semântica baseada em álgebra inicial. Sua sintaxe não é muito simples (parâmetros, importação, exportação, variáveis com atributos, etc.). Usa diagramas estruturados para representar cada módulo. Não tem OO. Usada somente para especificar sistemas seqüenciais.
- Cada especificação LOTOS tem duas componentes, cada uma em uma linguagem diferente: LOTOS Estático usando ACT ONE e LOTOS Dinâmico usando CSP e CCS. Foi projetada para aplicações específicas (serviços e protocolos do modelo de Referência OSI, Open Systems Interconection). Não tem OO.

- ACT ONE: sintaxe muito complicada (conceitos de união, renomeação, composição, produto, atualização e modularização, muitas palavras-chave), semântica baseada em álgebras e functores.

C. **O Método de Desenvolvimento de Software**

Foi apresentado uma seqüência de tarefas que deve-se realizar desde a apresentação de um problema até a especificação do sistema usando a linguagem LOP.

Finalmente, como uma extensão ao trabalho realizado, gostaria de propor algumas tarefas que poderiam ser implementadas como parte de trabalhos de graduação ou de mestrado em Ciência da Computação:

- Construção de editores dirigidos por sintaxe, em qualquer sistema computacional (Unix, MS Windows, etc.) para a linguagem LOP.
- Construção de verificadores sintáticos.
- Adaptação do Larch Prover para especificações LOP.
- Construção de uma biblioteca básica de especificações LOP.

ANEXO A-1 CONCEITOS BÁSICOS

Nesta seção apresentamos algumas definições na sua maioria extraídas da Álgebra Universal e da Teoria de Conjuntos, que serão necessárias para entender melhor os conceitos relacionados a linguagem a ser definida. Obviamente, são usados algumas expressões matemáticas que supõe-se conhecidas para leitor interessado em especificação formal.

A-1.1 Signaturas

Um **sort** é um identificador (nome ou símbolo) de um objeto real ou de um domínio de valores. Sorts são usados em linguagens de especificação algébrica para representar conjuntos de valores e corresponde a TIPO nas linguagens de programação (C, Pascal, etc.).

Ling. Programação :	x : Book $\longleftarrow$ x é do *tipo* Book.
Ling. Algébrica :	$x \in$ Book $\longleftarrow$ x é de *sort* Book.

Exemplo A-1.1

São sorts: Stack, S, FornProd, Emp, etc. onde:

Stack identifica pilhas.

S identifica um conjunto qualquer.

FornProd identifica um conjunto de fornecedores de um produto

Emp identifica uma empresa.

Dado um conjunto S $= \{s_1, \ldots, s_n\}$ de sorts, um *símbolo de operador* ou simplesmente *operador*, é uma $(k+2)$-tupla, $k \geq 0$, denotado por:

$$f : s_{i_1}, \ldots, s_{i_k} \longrightarrow s_{i_{k+1}}$$

onde $(s_{i_1}, \ldots, s_{i_k}, s_{i_{k+1}})$ é uma $(k+1)$-tupla de sorts, chamada *funcionalidade* de f.

Se $< s_1, s_2, \ldots, s_n, s_m >$ é a funcionalidade de f, então f pode ser escrito como :

$f : s_1 \times s_2 \times \ldots \times s_n \longrightarrow s_m$ ou

$f : S^* \longrightarrow s_m$ ou

$f : S^n \longrightarrow s_m$

$S^* = S^n = w = s_1 \times s_2 \times \ldots \times s_n$ é chamado de *aridade* de f. O sort s_m é chamado de *co-aridade* de f.

$$f : \overbrace{\underbrace{s_1 \times s_2 \times \ldots \times s_n}_{aridade\ de\ f} \longrightarrow \underbrace{s_m}_{co-aridade}}^{funcionalidade\ de\ f}$$

Dado um conjunto S = $\{s_1, \ldots, s_n\}$ de sorts, e um conjunto $\{f_1, \ldots, f_m\}$ de operadores, uma *signatura S-sortida*[1], denotada por

$\Sigma =< S, F >$

ou $Sig =< S, F >$,

é o conjunto

$\{\underbrace{s_1, \ldots, s_n}_{S}, \underbrace{f_1, \ldots, f_m}_{F}\}$,

para $n \geq 1$ e $m \geq 1$,

Se Sig = < S, F > e existe Sig1 = <S1,F1> tal que S1⊆S e F1⊆F, então Sig1 é chamada uma *subsignatura* de Sig e, escreve-se Sig1⊆Sig.

Exemplo A-1.2

Consideremos a signatura

SoftwareStore = <S, F>=< { $s_1, s_2, \ldots, s_{13}$}, $\{f_1, f_2, f_3\}$ >

onde:

S = { OrderNo, OrderData, CustomerName, Address, ChargeNo,
SoftwareNo, Quantity, Title, Author, UnitPrice, Data,
List, Info }

F = { *order-data, order-list, software-info* }

[1]A palavra *signature* do inglês não tem tradução bemdefinida em português, motivo pelo qual, adotamos nossa tradução particular: *signatura*.

As seguintes signaturas são subsignaturas de SoftwareStore:

$$<\text{S1, F1}>=<\{s_1,\ldots,s_6\},\{f_1\}>,$$

$$<\text{S2, F2}>=<\{s_1,s_7,s_8,s_9\},\{f_2\}> \text{ e}$$

$$<\text{S3, F3}>=<\{s_7,s_{10},s_{11},s_{12},s_{13}\},\{f_3\}>$$

onde:

S1 = { OrderNo, Orderdata, CustomerName, Address, ChargeNo, Data }

F1 = { *order-data* }

S2 = { OrderNo, SoftwareNo, Quantity, List }

F2 = { *order-list* }

S3 = { SoftwareNo, Title, Author, UnitPrice, Info }

F3 = { *software-info* }

Outra maneira de representar estas subsignaturas é a seguinte:

$<\text{S1, F1}> = f_1 : s_1 \times \ldots \times s_5 \longrightarrow s_{11}$

= *order-data* : {OrderNo, OrderData, CustomerName,
Address, ChargeNo } $\longrightarrow$ Data

$<\text{S2, F2}> = f_2 : s_1 \times s_7 \times s_8 \longrightarrow s_{12}$

= *order-list* : {OrderNo, SoftwareNo, Quantity} $\longrightarrow$ List

$<\text{S3, F3}> = f_3 : s_7 \times s_{10} \times s_{11} \times s_{12} \longrightarrow s_{13}$

= *software-info* : {SoftwareNo, Title, Author, UnitPrice} $\longrightarrow$ Info

Logo a signatura

$$\text{SoftwareStore} = <\{s_1,s_2,\ldots,s_{13}\},\{f_1,f_2,f_3\}>$$

poderá ser denotada usando a notação de funções:

$$f_1 : s_1 \times \ldots \longrightarrow s_{11}$$

$$f_2 : s_1 \times \ldots \longrightarrow s_{12}$$

$$f_3 : s_7 \times \ldots \longrightarrow s_{13}$$

onde cada função representa uma subsignatura (ou uma signatura). Cada função f : D $\longrightarrow$ R induz a sua signatura:

$$<\{\text{D, R}\},\{f\}>.$$

Uma *Signatura Sortida-Ordenada* é uma tupla

$$\textbf{SigSO} = < \text{S}, \mathcal{R}, \text{F} >$$

onde <S,F> é uma signatura S-sortida e (S,$\mathcal{R}$) é um conjunto parcialmente ordenado ($\mathcal{R}$ é reflexiva, transitiva e antisimétrica).

Seja Sig = <S, F> uma signatura. Um conjunto multisortido (ou S-sortido) de *variáveis* X para Sig, é uma família de conjuntos disjuntos (dois a dois) de símbolos X_s, um para cada sort $s \in$S.

A-1.2 Termos e Fórmulas

Seja Sig = <S, F> uma signatura S-sortida e

$$\text{X} = \{Xs_1, Xs_2, ..., Xs_m\}$$

um conjunto S-sortido de variáveis. Um conjunto de *Sig(X)-termos* bem-formados ou simplesmente **X-termos**, denotado por $\text{T}_{Sig}(\text{X})$, é definido indutivamente como o menor conjunto (com respeito à inclusão) que possui as seguintes propriedades:

- Cada variável $x \in$Xs, $s \in$S, é um X-termo de sort s,
- Para cada símbolo de função f com f:{} $\longrightarrow s_m$, $f()$ é um X-termo de sort s_m, onde $f \in$F.
- Se f é símbolo de função tal que

  $$f : s_1 \times s_2 \times \cdots \times s_n \longrightarrow s_m \ , \quad f \in \text{F},$$

  $n \geq 1$, e se $t_1, t_2, \ldots, t_n$ são X-termos de sort $s_1, s_2, \cdots, s_n$ respectivamente, então $f(t_1, t_2, \ldots, t_n)$ é um X-termo de sort s_m.

Seja t um X-termo, então **Var**(t) denota o conjunto de variáveis que ocorrem em t .

Um Sig(X)-termo t sem variáveis (Var(t) vazio) recebe o nome de *Sig-termo*, ou simplesmente, **termo** (*Sig-sentença , fórmula fechada, termo básico, termo canônico, etc.*).

Exemplo A-1.3

Seja Sig = <{ Nat, Bool }, { *zero, succ, equal* } > onde:

$zero$: {} $\longrightarrow$ Nat

$succ$: Nat $\longrightarrow$ Nat

$equal$: Nat, Nat $\longrightarrow$ Bool

- Se $x \in X_{Nat}$, $b \in X_{Bool}$ são variáveis, então x é um X-termo e b também é um X-termo (parte a).).

- $zero()$ é um X-termo de sort Nat.

- Se $x \in X_{Nat}$ então:

  $t_1 = succ(x)$ é um X-termo de sort Nat.

  $t_2 = succ(zero())$ também é um X-termo de sort Nat.

  $t_3 = succ(t_1) = succ(succ(x))$ é um X-termo de sort Nat.

  $t_4 = succ(t_2) = succ(succ(zero()))$ é um X-termo .

- Se $t_1,\ t_2 \in X_{Nat}$ então:

  $equal(t_1, t_2)$ é um X-termo de sort Bool.

  $equal(zero(), t_1)$ é um X-termo de sort Bool.

  $equal(succ(t_1), succ(zero()))$ também é um X-termo de sort Bool.

Seja Sig uma signatura e $\ll$ a relação

$$\ll : \mathrm{T}_{Sig}(\mathrm{X}) \longrightarrow \mathrm{T}_{Sig}(\mathrm{X})$$

definida indutivamente por:

- $t \ll t$ para todo Sig(X)-termo t.

- $t_i \ll f(t_1, \ldots, t_i, \ldots, t_n)$ para algum $i \in \{1, \ldots, n\}$ e $f(t_1, \ldots, t_i, \ldots, t_n)$ $\in$ Sig(X)-termos.

- Se $t_1 \ll t_2$ e $t_2 \ll t_3$ então $t_1 \ll t_3$ para todo Sig(X)-termo t_1, t_2 e t_3.

Então, um termo t_1 é chamado um *subtermo* do termo t_2 se $t_1 \ll t_2$.

Um termo t é *linear* se, e somente se, cada variável x em Var(t) ocorre somente uma vez.

Seja Sig = <S , F> uma signatura e X um conjunto S-sortido de variáveis. Uma *Sig(X)-fórmula atômica* é uma das duas expressões seguintes:
a) $t_1 == t_2$, onde t_1, t_2 são Sig(X)-termos do mesmo sort.
b) $D(t)$, onde D e um predicado definido ("definedness") e t é um Sig(X)-termo:

$$\mathrm{D} : \mathrm{T}_{Sig}(\mathrm{X}) \longrightarrow \{\ true(), false()\ \}$$

A expressão "$t_1 == t_2''$ é chamada *Sig(X)-equação*, ou simplesmente *equação.*

Seja Sig = <S , F>. Uma *Sig(X)-fórmula* ou simplesmente **fórmula**, é definida indutivamente como uma fórmula atômica ou uma das seguintes expressões: $\neg fbf$, $fbf_1 \vee fbf_2$, $fbf_1 \wedge fbf_2$, $fbf_1 \Longrightarrow fbf_2$, $\forall s_x : fbf$, $\exists s_x : fbf$ onde s é um sort de S, x é uma variável de Xs, e fbf, fbf_1 e fbf_2 são por sua vez fórmulas. As fórmulas são chamadas também de *Axiomas.* Em geral um axioma é representado por $\forall X(t_1 == t_2)$, onde t_1 e t_2 são fórmulas do mesmo sort, e X é um conjunto finito de variáveis. A fórmula $\neg fbf$ é equivalente a $\neg fbf == true$.

Uma *equação condicional* (fórmula condicional, fórmula condicional positiva, etc.) é um axioma da forma

$$f_1 \wedge \cdots \wedge f_n \Longrightarrow f_m$$

onde $f_1, \ldots, f_n, f_m$ são fórmulas. Observe que a equação condicional acima, é equivalente a $\neg(f_1 \wedge \cdots \wedge f_n) \vee f_m == true()$, pois, $(a \Longrightarrow b) = (\neg a \vee b)$.

A *extensão* (campo ou escopo) de um quantificador nas fórmulas $\exists\ xF$ ou $\forall xF$ é a fórmula F. Por exemplo,

$F_1 = \exists\ y\ (p(x) \longrightarrow\ q(y))$

extensão de $\exists$ em F_1 : $(p(x) \longrightarrow q(y))$

$F_2 = \forall\ x\ (y \wedge q(x) \longrightarrow r(x,y))$

extensão de $\forall$ em F_2 : $(y \wedge q(x) \longrightarrow r(x,y))$

Uma ocorrência *ligada* ou *limitada* de uma variável numa fórmula é:
- uma ocorrência que segue imediatamente a um quantificador , ou
- uma ocorrência dentro da extensão de um quantificador que tem a mesma variável que acompanha ao quantificador.

Qualquer outra ocorrência de uma variável é chamada *livre.*

Exemplo A-1.4

F = $\forall x(0 \leq x \leq 2 \longrightarrow \exists y(3 \leq y \leq 7 : y = x^2 + 3))$

x e y tem três ocorrências limitadas em F .

G = $\forall bl, br \in BTree : (left(make(bl, item, br)) = bl)$

bl tem três ocorrências limitadas em G

br tem duas ocorrências limitadas em G

item tem uma ocorrência livre em G

H = $\forall f \in$EmpX (nome(f) = "Peter Rhew' $\longrightarrow$ cod(f) = '1831-91')

f tem duas ocorrências limitadas em H.

Uma variável x é *livre numa fórmula* F se, e somente se, existe pelo menos uma ocorrência livre de x em F. Por exemplo,

F = $\forall x((x \vee y) \longrightarrow \exists y(y \vee x))$

y é livre em F mais x não é livre em F.

G = $\forall s(Pop(Push(s,x)) = s)$

x é livre em G mais s não é livre em G.

O *fecho universal* de uma fórmula P é a expressão

$$\forall x_1, \ldots, \forall x_n \mathrm{P}$$

onde $x_1, \ldots, x_n$ são todas variáveis livres em P. Assim, se P é a fórmula

$$isempty(append(q, e)) = true()$$

com q e e como variáveis livres em P, então o fecho universal de P é

$$\forall q \forall e(\ isempty(append(q, e)) = true()\ )$$

A *substituição* de uma fórmula f por uma variável x em uma fórmula P, denotada por $P[x \leftarrow f]$, é o resultado de substituir simultaneamente cada ocorrência livre de x em P pela fórmula f, depois de renomear as variáveis ligadas necessárias de modo a evitar a captura de variáveis livres em f.

Exemplo A-1.5

- A substituição $rest(insert(s, e))[s \leftarrow new()]$ produz a fórmula $rest(insert(new(), e))$.
- Se $f = insert(z, e)$ então a substituição

$$rest(insert(s, e))[s \leftarrow f]$$

produz a fórmula

$$rest(insert(insert(z, k), e)$$

onde a variável e de f foi renomeada para k.

Uma *fórmula fechada* (ou fórmula constante) é uma fórmula sem variáveis; i.e., uma fórmula onde cada ocorrência da variável x de sort s, aparece (implicita ou explicitamente) sem os quantificadores universal $\forall$ e existencial $\exists$.

Uma signatura Sig = <S , F> *estendida* por meio de um conjunto finito de axiomas Ax é chamada uma *Apresentação* e denotada por

Ap = <S , F , Ax>

A-1.3 Álgebras

Informalmente, uma Álgebra Multi-sortida (MSA) é uma estrutura matemática formada por :

a) uma família de conjuntos de objetos chamados *termos*, e

b) um conjunto de funções, com argumentos e valores nesses objetos, chamadas *operações.*

Exemplo A-1.6

A pilha de números naturais NS = < SSet, SFunc > é uma álgebra onde

SSet = { Stack, Natural, Bool }

SFunc = { *push, pop, top, isnewstack* }

Seja Sig = <S , F> uma signatura S-sortida e X um conjunto de variáveis S-sortidas. Uma *Sig(X)-álgebra multisortida (MSA)* A ou simplesmente *álgebra* A, é um par definido por:

- uma coleção de conjuntos não vazios $\{As_i\}$, para cada $s_i \in$S. O conjunto As_i é chamado o *carregador* (carrier) de sort s_i.

- uma coleção $\{ Af_j \}$ de funções entre os conjuntos As_i, onde

$$Af_j : As_k \times As_{k+1} \times \cdots \longrightarrow As_m$$

chamada *Operação de A rotulada por f*, para cada $f_j \in$F, $n \geq 0$,$j, k \in$**N**

$$f_j : s_k \times s_{k+1} \times \cdots \longrightarrow s_m$$

Os chamados *objetos* de uma álgebra são os carregadores As_i e as operações Af_j.

Notação:

$$A = < \{As_i\} , \{Af_j\} >$$

Se SigSO = <S , R, F> é uma signatura sortida-ordenada, então a SigSO-álgebra A é chamada *Álgebra Sortida Ordenada* (OSA)

Dependendo das funções de $\{Af_j\}$ serem parciais ou totais, uma álgebra A será *Álgebra Parcial* ou *Álgebra Total*, respectivamente.

Se A é uma Sig-álgebra, então diz-se que A é *descrita pela signatura* Sig. Se A é descrita pela signatura Sig de uma apresentação Ap, diz-se que A é *descrita pela apresentação* Ap (Ap-álgebra).

Duas álgebras com a mesma signatura, são ditas ser *Similares.*

Alg(Sig) denota a classe de todas as Sig-álgebras .

Uma *Álgebra de Palavras P* descrita pela signatura Sig = <S, F>, é uma álgebra (como definido acima) onde os objetos do conjunto As_j, para cada sort s_j de S, são Sig-termos (termos sem variáveis) construídos em um número finito de passos usando somente os símbolos de funções de F e considerados como cadeias de caracteres. Estes Sig-termos são chamados *termos sem variáveis, termos livres de variáveis* ou *termos básicos* . Uma álgebra de palavras é chamada também *Universo de Herbrand* ou *Álgebra de termos* (do inglês 'term algebra').

Dadas duas Sig-álgebras A e B, uma função

$$h : A \longrightarrow B$$

do conjunto de objetos de A para o conjunto de objetos de B, é chamado um *homomorfismo* se, e somente se, o comportamento das operações (a estrutura da álgebra) é preservada [van 89a]:

$$h(As_i) = B_{h(s_i)}$$
$$h(Af_j) = B_{h(f_j)}$$

Seja K$\subseteq$Alg(Sig) uma classe de Sig-álgebras. Uma álgebra I$\in$K é chamada **Inicial** em K, se para toda A$\in$K, existe *exatamente* um homomorfismo de I para A. Uma álgebra F$\in$K, é **Final** ou *terminal* em K, se para toda A$\in$K, existe *pelo menos um* homomorfismo $h : A \longrightarrow F$.

Equivalentemente, uma álgebra I é *Inicial* em K se, I é o limite superior mínimo de K. Uma álgebra F é *Final* em K se F é o limite inferior máximo de K. Em ambos casos, a preordem R em K é induzida pela existência de homomorfismos, i.e., (A,B)$\in R$ em K se existe $h : A \longrightarrow B$.

Uma *álgebra inicial* definida por uma apresentação é o Universo de Herbrand que, satisfazendo os axiomas dados, tem o maior número possível de objetos.

Uma *álgebra final* definida por uma apresentação é o Universo de Herbrand que, satisfazendo os axiomas dados, tem o menor número possível de objetos.

Na filosofia de uma *álgebra inicial*, dois termos livres de variáveis denotam dois objetos diferentes, a menos que possa ser provado, a partir dos axiomas dados, que tais termos denotam o mesmo objeto.

Em uma *álgebra final*, dois termos sem variáveis (do mesmo sort) sempre denotarão o mesmo objeto, a menos que, a partir dos axiomas dados, possa ser provado que os dois termos denotam objetos diferentes.

Seja A uma Σ-álgebra e seja Y um subconjunto de A. Então a *subálgebra gerada por* Y é a menor subálgebra de A que contém Y. Uma álgebra gerada por Y é denotada por $\langle Y \rangle$

$$\langle Y \rangle = \bigcap \{B / Y \subseteq B, \quad B \text{ subálgebra de } A\}$$

A-1.4 Teorias

Uma *teoria de primeira ordem* é uma tupla $< \alpha, L, \Lambda, \Re >$ onde α é um alfabeto (conjunto de símbolos), L é uma linguagem de primeira ordem, Λ um conjunto de axiomas, e $\Re$ um conjunto de regras de inferência. Uma *linguagem de primeira ordem* dada por um alfabeto é o conjunto de todas as fórmulas (bem-formadas) construídas a partir dos símbolos do alfabeto. Os *axiomas* é um subconjunto da linguagem de primeira ordem.

Outra definição equivalente de uma teoria de primeira ordem é a seguinte: uma *teoria Th* é um conjunto de fórmulas F (em algum sistema lingüístico), fechado sob uma relação de derivabilidade $\mathcal{R}$.

$$Th =< F, \mathcal{R} >$$

Uma *relação de derivabilidade* $\mathcal{R}$ sobre um conjunto F, é uma relação binária

$$\mathcal{R} \subset Pot(F) \times F$$

de modo que, para dois subconjuntos $G, G' \subset$ F, e duas fórmulas $a, b \in$ F, temos:

- Se $a \in G$ então $(G, a) \in \mathcal{R}$.
- Se $G \subseteq G'$ e $(G, a) \in \mathcal{R}$ então $(G', a) \in \mathcal{R}$.
- $(G, a) \in \mathcal{R}$ e $(G \cup \{a\}, b) \in \mathcal{R}$ então $(G \cup G', b) \in \mathcal{R}$.
- Se $(G, a) \in \mathcal{R}$ então $(G', a) \in \mathcal{R}$ para algum subconjunto finito G' de G.
- Se $(G, a) \in \mathcal{R}$ então renomeação consistente de variáveis proposicionais em a e G não modificarão a relação de deducibilidade.

Informalmente, $(G, a) \in \mathcal{R}$ significa que a fórmula a é uma conseqüência ou pode ser derivada das fórmulas contidas em G. Geralmente, a relação $\mathcal{R}$ é denotada também pelos símbolos $\vdash$ ou $\models$.

Uma teoria $< F, \mathcal{R} >$ é *consistente*, se e somente se, para uma fórmula $a \in F$, se $(F, a) \in \mathcal{R}$ então $(F, \neg a) \notin \mathcal{R}$.

Uma teoria $Th_1 =< F_1, \mathcal{R}_1 >$ é uma *extensão conservativa* de outra teoria $Th =< F, \mathcal{R} >$, denotado por $Th \ll Th_1$, se e somente se, $F \subseteq F_1$ e para qualquer fórmula $a \in F$, se $(F_1, a) \in \mathcal{R}_1$ então $(F, a) \in \mathcal{R}$.

A-1.5 Indução

Seja U o conjunto chamado *Universo*, B um subconjunto de U e K um conjunto de relações $r \subseteq U^n \times U$ $(n \geq 1)$ chamado *conjunto construtor*. Um

conjunto A é dito ser *definido indutivamente* por B e K, quando A é o menor (com respeito à inclusão) de todos os conjuntos S de U para os quais valem asseguintes duas condições:

- B$\subseteq$S.
- $r(S^n) \subseteq S$ para todo $r \in$K, i.é., se $a_1, \ldots, a_n \in S$ e $((a_1, \ldots, a_n), a) \in r$ para algum construtor $r \in$K, então $a \in$S.

Sejam U, B e K como na definição anterior. Uma sequência $u_1, \ldots, u_m$ ($m \geq 1$) de elementos de U com $u_m = u$ é chamada uma *sequência de construção* para u, a partir de B e K, quando para cada $i = 1, \ldots, m$ tem-se:

- $u_i \in$B, ou
- existe um construtor $r \subseteq U^n \times U$ de K e $i_1, \ldots, i_n < i$, tal que, $((u_{i1}, \ldots, u_{in}), u_i) \in r$.

O conjunto A$\subseteq$U é dito ser **definido indutivamente** por B e K, quando A está formado por todos os elementos $u \in$U, para os quais existe uma seqüência de construção.

As provas dos teoremas, que são enunciados a seguir, podem ser encontrados em [WEC 92].

Teorema (Princípio de Indução Estrutural). Seja A uma Σ-álgebra gerada por X. Para provar que uma propriedade $\mathcal{P}$ vale para todos os elementos de A, é suficiente provar que:
a). $\mathcal{P}$ vale para todos os elementos de X.
b). se $\mathcal{P}$ vale para qualquer $a_1, \ldots, a_n \in$ A, então $\mathcal{P}$ vale para a operação $A_f(a_1, \ldots, a_n)$ para todo $f \in \Sigma$, $n \in$Nat.

Teorema (Princípio de Indução Noetheriano). Seja $\mathcal{R}$ uma relação noetheriana sobre um conjunto A. Uma propriedade $\mathcal{P}$ vale para qualquer elemento a de

A se $\mathcal{P}$ vale para todos os x em A com $a\mathcal{R}x$, isto é, se

$$\forall a \in A\Big(\forall x \in A\big(a\mathcal{R}x \Rightarrow \mathcal{P}(x)\big) \Rightarrow \mathcal{P}(a)\Big)$$

então $\mathcal{P}$ vale para todo a em A.

Teorema (Princípio de Recursão Algébrica Finito). Seja Σ uma signatura e X um conjunto de variáveis. Cada mapeamento f de X para a Σ-álgebra A admite uma única extensão homomórfica

$$f^{\#} : \mathrm{T}_{\Sigma}(\mathrm{X}) \longrightarrow \mathrm{A} .$$

A-1.6 Métodos e Linguagens

- **Método:**

  " Um *método* é uma coleção de *procedimentos*, isto é, diretivas para usar, em uma determinada seqüência, um número de *técnicas* e *ferramentas*, de modo que um problema dado é resolvido com eficiência (e quando está sendo resolvido, é resolvido corretamente).

  Um *método* é um procedimento com um critério de seleção para escolher e usar um número de *técnicas* e *ferramentas* para obter eficientemente a construção de um produto eficiente.

  Metodologia é o estudo de conhecimento comparativo sobre uma familia de métodos. "([BJO 88], vol. I).

- **Método Formal:**

  Métodos Formais usados no desenvolvimento de sistemas de computador são técnicas baseadas matematicamente para descrever propriedades do sistema.

  Um método é *formal*, se este tem uma sólida base matemática, tipicamente, dada por uma linguagem de especificação formal.[WIN 90]

- **Especificação Formal:**

Uma *especificação* é a descrição de um conjunto especificando. Um *conjunto especificando* é o conjunto de objetos que formam a especificação. Por exemplo, uma gramática é a especificação de uma linguagem formal, o conjunto especificando de uma gramática é a linguagem (o conjunto de sentenças), um programa é a especificação de um conjunto de computações, o conjunto especificando de um programa é o conjunto de computações que pode-se obter ao executar o programa.

Uma *especificação* é *formal* se é escrita completamente em uma linguagem com uma sintaxe e semântica definidas explicitamente e com precisão ([WIN 90]).

Uma *especificação*, um projeto ou um desenvolvimento (especificação + projeto) é *formal* se todos os documentos e todas as obrigações de prova implicadas, são expressadas formalmente e se todas as obrigações de prova são formalmente descarregadas ([BJO 88], vol.I)

- **Linguagens de Especificação Formal:**

Uma *linguagem de especificação* é uma notação para apresentar especificações. Uma *linguagem de especificação formal* é uma notação não ambigua, bem definida e precisa.

Uma *linguagem de especificação formal* L é uma tupla <Syn, Sem, Sat> onde:

Syn é um conjunto chamado *domínio sintático* da linguagem.

Sem é um conjunto chamado *domínio semântico* da linguagem.

Sat é uma relação chamada *relação de satisfação* (Sat $\subseteq$ Syn $\times$ Sem).

Se (syn,sem)$\in$Sat então:

syn é uma *especificação* de sem.

sem é um *especificando* de syn.

Informalmente, uma *linguagem de especificação formal* fornece uma notação (domínio sintático), um universo de objetos (domínio semântico) e uma regra precisa que define quais objetos satisfazem cada especificação.

Um **método** determina o *que* uma especificação deve dizer. Uma **linguagem** determina em detalhes *como* a especificação é dita ([LAM 89], p.32) .

ANEXO A-2 CLASSES LOP

Apresentamos aqui alguns módulos LOP que foram usados implicitamente ou explicitamente nos exemplos dos capítulos anteriores e que junto com alguns daqueles módulos, por serem de interesse geral, poderiam integrar uma biblioteca compartilhada LOP.

BOOL : **class**

introduces

$$
\begin{array}{rcll}
true & : & \{\} & \longrightarrow \text{Bool} \\
false & : & \{\} & \longrightarrow \text{Bool} \\
\neg\_ & : & \text{Bool} & \longrightarrow \text{Bool} \\
\_\wedge\_ & : & \text{Bool, Bool} & \longrightarrow \text{Bool} \\
\_\vee\_ & : & \text{Bool, Bool} & \longrightarrow \text{Bool} \\
\Rightarrow & : & \text{Bool, Bool} & \longrightarrow \text{Bool}
\end{array}
$$

asserts

Bool **generated by** $true$, $false$

for all b: Bool

$\neg true() == false()$

$\neg false() == true()$

$true() \wedge b == b$

$false() \wedge b == false()$

$true() \vee b == true()$

$false() \vee b == b$

$true() \Rightarrow b == b$

$false() \Rightarrow b == true()$

IF_THEN_EQ : **class**

introduces

$$\_ = \_ \ : \ \text{S, S} \longrightarrow \text{Bool}$$

$$\_ \neq \_ \ : \ \text{S, S} \longrightarrow \text{Bool}$$

$$if\_then\_else\_ \ : \ \text{Bool, S, S} \longrightarrow$$

asserts

S **partitioned by** $=$

for all x, y, z: S

$(x = x) == true()$

$(x = y) == (y = x)$

$(x = y \ \wedge \ y = z) \Rightarrow (x = z) == true()$

$(x \neq y) == \neg(x = y)$

$if \ true() \ then \ x \ else \ y == x$

$if \ false() \ then \ x \ else \ y == y$

CONTAINER_OPS : **class**

introduces

$$newq \ : \{\ \} \longrightarrow \text{C}$$

$$insert \ : \text{C, E} \longrightarrow \text{C}$$

asserts

C **generated by** *new*, *insert*

CONTAINER(C,E) : **class**

introduces

$$newq \ : \{\ \} \longrightarrow \text{C}$$

$$insert \ : \text{C, E} \longrightarrow \text{C}$$

asserts

C **generated by** *new*, *insert*

LINEARCONTAINER(C,E) : **class**

aggregation of CONTAINER_OPS

introduces

$first$: C $\longrightarrow$ E

$rest$: C $\longrightarrow$ C

$isempty$: C $\longrightarrow$ Bool

asserts

C **partitioned by** *first, rest, isempty*

for all c: C, e: E

$isempty(new()) == true()$ – $LC1$ –

$isempty(append(c, e)) == false()$ – $LC2$ –

$first(insert(c, e)) ==$ **if** $isempty(c)$ – $LC3$ –

then e

else $first(c)$

$rest(insert(c, e)) ==$ **if** $isempty(c)$ – $LC4$ –

then $new()$

else $insert(rest(c), e)$

implies converts *isempty*

DIRECTIONS : **class**

introduces

$north, east, south, west$: $\{\ \}$ $\longrightarrow$ Direction

$turnleft, turnright, opposite$: Direction $\longrightarrow$ Direction

asserts

$turnleft(north()) == west()$

$turnleft(south()) == east()$

$turnleft(east()) == north()$

$turnleft(west()) == south()$

$turnright(north()) == east()$

$turnright(south()) == west()$

$turnright(east()) == south()$

$turnright(west()) == north()$

$opposite(north()) == south()$

$opposite(south()) == north()$

$opposite(east()) == west()$

$opposite(west()) == east()$

AC : **class**

aggregation of ASSOCIATIVE, COMMUTATIVE(T **for** Range)

ASSOCIATIVE : **class**

introduces

$\_ \circ \_ : T, T \longrightarrow T$

asserts for all x, y, z: T

$(x \circ y) \circ z == x \circ (y \circ z)$

COMMUTATIVE : **class**

introduces

$\_ \circ \_ : T, T \longrightarrow Range$

asserts for all x, y: T

$x \circ y == y \circ x$

DERIVED_ORDERS(T) : **class**

introduces

$\leq : T, T \longrightarrow Bool$

$\geq : T, T \longrightarrow Bool$

$< : T, T \longrightarrow Bool$

$> : T, T \longrightarrow Bool$

asserts

for all $x, y : T$

$x \leq y == (x < y) \vee (x = y)$

$x \geq y == y \leq x$

$x < y == (x \leq y) \wedge \neg(y \leq x)$

$x > y == y < x$

implies

converts $\geq, <, >$

converts $\leq, <, >$

converts $\leq, \geq, >$

converts $\leq, \geq, <$

SET(C, E) : **class**

aggregation of CARDINAL

DERIVED_ORDERS(C, $\subseteq$ **for** $\leq$, *supseteq* **for** $\geq$,
$\subset$ **for** $<$, *supset* **for** $>$)

introduces

$\{\}$	:		$\longrightarrow$ C
insert	:	C,E	$\longrightarrow$ C
$\_\in\_$	:	E,C	$\longrightarrow$ Bool
$\_\neg\in$	:	E,C	$\longrightarrow$ Bool
$\{\_\}$	:	E	$\longrightarrow$ C
$\_\cup\_$	:	C, C	$\longrightarrow$ C
$\_\cap\_$	:	C, C	$\longrightarrow$ C
$\_-\_$	:	C, C	$\longrightarrow$ C
delete	:	C, E	$\longrightarrow$ C
isEmpty	:	C	$\longrightarrow$ Bool
size	:	C	$\longrightarrow$ Card

asserts

C **generated by** $\{\}, insert$

C **partitioned by** $\in$

for all s_1, s_2: C, e_1, e_2: E

$\neg(e_1 \in \{\}) == true()$

$e_1 \in insert(s_1, e_2) == (e_1 = e_2 \ \vee \ e_1 \in s_1)$

$e_1 \neg \in s_1 == \neg(e_1 \in s_1)$

$\{e_1\} == insert(\{\}, e_1)$

$e_1 \in (s_1 \cup s_2) == (e_1 \in s_1) \vee (e_1 \in s_2)$

$e_1 \in (s_1 \cap s_2) == (e_1 \in s_1) \wedge (e_1 \in s_2)$

$e_1 \in (s_1 - -s_2) == (e_1 \in s_1) \wedge (e_1 \neg \in s_2)$

$e_1 \in delete(s_1, e_2) == (e_1 \in s_1) \wedge (e_1 \neq e_2)$

$isEmpty(\{\}) == true()$

$\neg isEmpty(insert(s_1, e_1)) == true()$

$size(\{\}) == 0$

$size(insert(s_1, e_1)) ==$ **if** $e_1 \in s_1$

then $size(s_1)$

else $succ(size(s_1))$

implies

$AC(\cup, C)$

$AC(\cap, C)$

$PARTIAL\_ORDER(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq, \subset$ **for** $<, \supset$ **for** $>)$

$CONTAINER(\{\} fornew)$

C**generatedby**$\{\}, \{\_\}, \cup$

forall$s : C, e_1, e_2 : E$

$insert(insert(s, e_1), e_1) == insert(s, e_1)$

$insert(insert(s, e_1), e_2) == insert(insert(s, e_2), e_1)$

converts $\in, \neg \in, \{\_\_\}, \cup, \cap, delete, isEmpty, size, \subseteq, \supseteq, \subset, \supset$

NATURAL : **class**

specialization of CARDINAL(Nat **for** Card)

implies

AbelianSemigroup(Nat,+)

AbelianSemigroup(Nat,*)

Distributive(Nat **for** T)

BIBLIOGRAFIA

[ARU 92] ARUN-KUMAR, S.; HENNESSY, M. An efficiency preorder for processes. **Acta Informatica**, Berlin, v.29, n.8, p.737-760, Dec. 1992.

[AST 89] ASTESIANO,E.; GIOVINI,A.; REGGIO,G. et al. An Integrated Algebraic Approach to the Specification of Data Types, Processes and Objects. In: WIRSING,M.; BERGSTRA,J.A. (Eds.). **Algebraic Methods**: Theory, Tools and Applications. Berlin: Springer-Verlag, 1989. p.91-116. (Lecture Notes in Computer Science, 394).

[AST 91] ASTESIANO,E.; REGGIO,G. **A Structural Approach to the Formal Modelization and Specification of Concurrent Systems**. Genova: Formal Methods Group, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1991. (Technical Report 0).

[AST 93] ASTESIANO,E.; REGGIO,G. Algebraic Specification of Concurrency. In: **Recent trends in Data Type Specifications**. Berlin: Springer-Verlag, 1993. (Lecture Notes in Computer Science, 655).

[AST 93a] ASTESIANO,E.; REGGIO,G. A Metalanguage for the Formal Requirement Specification of Reactive Systems. In: INTERNATIONAL SYMPOSIUM OF FORMAL METHODS EUROPE - FME'93, 1., April 1993, Odense, Denmark. **Proceedings....** Berlin: Springer-Verlag, 1993. (Lecture Notes in Computer Science, 670).

[AST 93b] ASTESIANO,E.; REGGIO,G. **Specifying Reactive Systems by Abstract Events**. Genova: Draft Report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Mar. 1993.

[BAE 90] BAETEN, J.C.M.; BERGSTRA, J. A. Process Algebra with zero and non-determinacy. In: CONCUR90, Amsterdam, 1990. **Proce-**

edings.... Berlin: Springer Verlag, 1990. p.83-98. (Lecture Notes in Computer Science, 458).

[BAE 91] BAETEN, J.C.M.; BERGSTRA, J. A. Asynchronous communication in real space process algebra, In: FORMAL TECHNIQUES IN REAL-TIME AND FAULT-TOLERANT SYSTEMS, Nijmegen, 1991. **Proceedings....** Berlin: Springer-Verlag, 1991. p. 473-492. (Lecture Notes in Computer Science, 571).

[BAE 92] BAETEN, J.C.M.; VAANDRAGER,F.W. An Algebra for process creation. **Acta Informatica**, Berlin, v.29, n.4, p.33-334, July 1992.

[BAE 92a] BAETEN, J.C.M.; BERGSTRA, J.A.; SMOLKA, S.A. Axiomatizing Probabilistic Processes: ACP with Generative Probabilities. In: INTERNATIONAL CONFERENCE ON CONCURRENCY THEORY - CONCUR'92, 3., Aug. 1992, Stony Brook. **Proceedings....** Berlin: Springer-Verlag, 1992. p. 472-485. (Lecture Notes in Computer Science, 630).

[BAI 89] BAILIN,S.C. An Object-Oriented Requirements Specification Method. **Communications of the ACM**, New York, v.32, n.5, p.608-623, May 1989.

[BER 84] BERGSTRA,J.A.; KLOP,J.W. Process Algebra for Synchronous Communication. **Information and Control**, Orlando, FL, v.60, n.1-3, Jan./Mar. 1994.

[BER 89] BERGSTRA,J.A.; KLOP,J.W. ACP_τ : A Universal Axiom System for Process Specification. In: WIRSING, M.; BERGSTRA,J.A. (Eds.). **Algebraic Methods** : Theory, Tools and Applications. Berlin: Springer-Verlag, 1989. p.447-463. (Lecture Notes in Computer Science, 394).

[BER 89a] BERGSTRA, J.A.; HEERING,J.; KLINT,P. **Algebraic Specification**. New York: Addison-Wesley, 1989.

[BER 90] BERGSTRA, J.A.; HEERING,J.; KLINT,P. Module Algebra. **Journal of the Association for Computing Machinery**, New York, v.37, n.2, p.335-372. April 1990.

[BJO 78] BJORNER, D.; JONES, C.B. **The Vienna Development Method:** The Meta Language. Berlin: Springer-Verlag, 1978. (Lecture Notes in Computer Science, 61).

[BJO 88] BJORNER, D. **Software Architectures and Programming Systems Design - The VDM Approach**. Lyngby: Departament of Computer Science, Technical University of Denmark, Oct. 1988.

[BOL 87] BOLOGNESI, T.; BRINKSMA, E. Introduction to the ISO Specification Language LOTOS. **Computer Networks and ISDN Systems**, Amsterdam, v.14, n.1, 1987.

[BOO 89] BOOCH, G. What Is and What Isn't Object-Oriented Design? **Am. Programmer**, [S.l.], v.2, n.7-8. p.14-21, Summer 1989.

[BRO 84] BROOKES, S.D.; HOARE, C.A.R.; ROSCOE,A.W. A Theory of Communicating Sequential Processes. **Journal of the ACM**, New York, v.31, n.3, p.560-599, July 1984.

[CAS 93] CASAIS, E.; LEWERENTZ, C. et al. **Formal Methods and Object-Orientation**. [S.l.:s.n.], 1993. Tutorial at TOOLS Europe, 1993, Versalles, France.

[CAS 92] CASTRO VERA, A. S. **Métodos e Linguagens de Especificação Formal**. Porto Alegre: CPGCC da UFRGS, Set. 1992.

[CAS 92a] CASTRO VERA, A. S. **Semântica Formal de Linguagens de Programação**. Porto Alegre: CPGCC da UFRGS, Set. 1992.

[CAS 94] CASTRO VERA, A. S. LOP: especificação algébrica + processos + objetos. In: CONFERENCIA LATINOAMERICANA DE INFORMÁTICA - PANEL'94, 20., Set. 1994. **Proceedings....** México, 1994. p.587-598.

[CAS 95] CASTRO VERA, A. S. An Introduction to LOP - An Unified Approach to Algebraic Specification, Process and Object-Orientation. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS RESEARCH, INFORMATICS AND CYBERNETICS - InterSymp'95, 5.; INFORMATION SYSTEMS ANALYSIS AND SYNTHESIS - ISAS'95, Aug. 1995. **Proceedings....** Baden-Baden, Germany, 1995.

[CHA 91] CHAMPEAUX, D. de, (moderator). Formal Techniques for OO Software Development (PANEL OOSPLA'91). **Sigplan Notices**, New York, v.26, n.11, p.166-170. Nov. 1991.

[CLE 88] CLERICI,S.; OREJAS,F. GSBL: An Algebraic Specification Language Based on Inheritance. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING - ECOOP'88, Oslo, Aug. 1988. **Proceedings....** [S.l.:s.n.], Aug. 1988. p.78-92. (Lecture Notes in Computer Science, 322).

[COA 92] COAD,P.; YOURDON,E. **Análise Baseada em Objetos**. São Paulo: Campus,1992.

[COS 92] COSTA, M. M. do C. Introdução à Lógica Modal Aplicada à Computação, In: ESCOLA DE COMPUTAÇÃO, 8., 1992, Gramado. **Anais....** Porto Alegre: Instituto de Informática da UFRGS, 1992.

[DAH 86] DAHL, O-J., LANGMYHR, D. F.; OWE, O. **Preliminary Report on the Specification and Programming Language ABEL**. Oslo: University of Oslo, 1986. (Research Report 106).

[DAN 88] DANFORTH,S.; TOMLINSON,Ch. Type Theories and Object-Oriented Programming. **ACM Computing Surveys**, New York, v.20, n.1, p.29-72, Mar. 1988.

[DUC 92] DUCOURNAU, R. et al. Monotonic Conflict Resolution Mechanisms for Inheritance. **ACM SIGPLAN NOTICES**, New York, v.27, n.10, p.16-24, Oct. 1992.

[DUK 90] DUKE, R. Formal Specification of Object-Oriented Systems. In: INTERNATIONAL CONFERENCE ON TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, 3., 1990, Nov. 28-30, Sidney. **Proceedings....** Sidney: TOOLS Pacific, 1990.

[EHR 85] EHRIG, H.; MAHR, B. **Fundamentals of Algebraic Specification**. Berlin: Springer-Verlag, 1985. 2v.

[FIC 92] FICHMAN, R.G.; KEMERER, Ch.F. Object-Oriented and Conventional Analysis and Design Methodologies - Comparison and Critique. **Computer**, New York, p.22-39, Oct. 1992.

[GAR 91] GARLAND, S.J.; GUTTAG, J.V. **A Guide to LP, The Larch Prover**. Palo Alto, CA : DEC/Systems ResearchCenter, Dec. 1991. (Report 82).

[GEO 90] GEORGE, Ch.; MILNE, R. **Specifying and refining concurrent systems - an example from the RAISE project**. Birkerod, Denmark: Computer Resources International A/S, Mar. 1990. (RAISE/STC/CWG/59/v1).

[GEO 92] GEORGE, Ch. **Specification and Development of an automatic train system**. Birkerod, Denmark: Computer Resources International A/S, Aug. 1992. (LACOS/CRI/CWG/38/v2).

[GEO 92a] GEORGE, Ch. **Specifying processes as subtypes.** Birkerod, Denmark: Computer Resources International A/S, Sep. 1992. (LACOS/CRI/CWG/42/v1).

[GER 92] GERMAN, S.M. Programming in a General Model of Synchronization. In: INTERNATIONAL CONFERENCE ON CONCURRENCY THEORY - CONCUR'92, 3., Aug. 1992, Stony Brook. **Proceedings....** Stony Brook: Springer-Verlag, 1992. p.534-549. (Lecture Notes in Computer Science, 630).

[GOG 78] GOGUEN, J. A.; THATCHER J.W. et al. An initial algebra approach to the specification, correctness and implementation of abstract data types. In: **Current Trends in Programming Methodology**. Englewoods Cliffs: Prentice-Hall, 1978. v.4.

[GOG 88] GOGUEN, J. A.; KIRCHNER, C. et al. An Introduction to OBJ-3. In: INTERNATIONAL WORKSHOP ON CONDITIONAL TERM REWRITING SYSTEMS, 1., June, 1988. **Proceedings....** Berlin: Springer-Verlag. (Lecture Notes in Computer Science, 308).

[GUT 78] GUTTAG, J. V. The Algebraic Specification of Abstract Data Types, **Acta Informatica**, Berlin, v.10, n.1, 1978.

[GUT 85] GUTTAG, J.V.; HORNING, J.J.; WING, J.M. **Larch in Five Easy Pieces**. Palo Alto, CA: DEC/SystemsResearch Center, July 1985. (Report 5).

[GUT 90] GUTTAG, J.V.; HORNING, J.J.; MODET,A. **Report on the Larch Shared Language : Version 2.3**. Palo Alto, CA: DEC/Systems Research Center, Apr. 1990. (Report 58).

[GUT 91] GUTTAG, J.V.; HORNING,J.J.: **Introduction to LCL, A Larch/C Interface Language**. Palo Alto, CA: DEC/Systems Research Center, July 1991. (Report 74).

[GUT 91a] GUTTAG, J.V.; HORNING,J.J. **The Larch Book - An LSL Handbook.** Palo Alto, CA: Digital Equipment Corporation. Version incompleta. Não publicada. Oct. 1991.

[HAL 87] HALBERT,D.C.; O'BRIEN, P.D. Using Types and Inheritance in Object-Oriented Programming. **IEEE Software**, Los Alamitos, CA, v.4, n.5, p.71-79. Sept. 1987.

[HAN 78] HANSEN, P.B. The Programming Language Concurrent Pascal. In: GRIES, D. (Ed.). **Programming Methodology**. New York: Springer-Verlag, 1978.

[HEN 90] HENDERSON-SELLERS, B. : Three Methodological Frameworks for Object-Oriented Systems Development. In: INTERNATIONAL CONFERENCE ON TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, 3., 1990, Nov. 28-30, Sidney. **Proceedings....** Sidney: TOOLS Pacific, 1990. p.118-131.

[HOA 85] HOARE,C.A.R. **Communicating Sequential Processes.** Englewoods Cliffs: Prentice-Hall, 1985.

[HOR 88] HORN, B. L. **An Introduction to Object-Oriented Programming, Inheritance and Method Combination**. Pittsburgh, PA: Dep. of Computer Science, Carnegie-Mellon University, Jan. 1988. (CMU-CS-87-127).

[IER 91] IERUSALIMSCHY, R. **The O=M Programming Language**. Rio de Janeiro: PUC, Dept. Informatica, 1991.

[IER 91a] IERUSALIMSCHY, R. **A Method for Object-Oriented Specifications with VDM.** Rio de Janeiro: PUC Dept.Informatica, 1991.

[JOH 88] JOHNSON, R.E.; FOOTE, B. Designing reusable classes. **Journal of Object-Oriented Programming**, New York, v.1, n.2, p.1,2,22-35, 1988.

[JON 86] JONES, C.B. **Systematic Software Development Using VDM**. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[JON 91] JONES,K.D.: **LM3 : A Larch Interface Language for Modula-3. A definition and Introduction. Version 1.0**. Palo Alto, CA: DEC/Systems Research Center, June 1991. (Report 72).

[KIR 88] KIRCHNER, C.; KIRCHNER, H. et al. Operational Semantics of OBJ3. In: INTERNATIONAL COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, 15., 1988, July 11-15, Tampere, Finland. **Proceedings....** Berlin: Springer-Verlag, 1988. p.287-301. (Lecture Notes in Computer Science, 317).

[KOR 90] KORSON,T.; McGREGOR, J.D. Understanding Object-Oriented: A Unifiying Paradigm. **Comunications of the ACM**, New York, v.33, n.9, p.40-60, Sept. 1990.

[LAM 89] LAMPORT, L. A Simple Approach to Specifying Concurrent Systems. **Communications of the ACM**, New York, v.32, n.1, Jan. 1989.

[LEH 88] LEHMANN, T.; LOECKX, J. The Specification Language of OBSCURE. In: RECENT TRENDS IN DATA TYPE SPECIFICATIONS, 1987, Sept. 1-4, Gullane, Scotland. **Proceedings....** Berlin: Springer-Verlag, 1988. p.131-153. (Lecture Notes in Computer Science, 332).

[LIM 92] LIMONGELLI, C.; TEMPERINI, M. Abstract Specifications of structures and methods in symbolic mathematical computation. **Theoretical Computer Science**, Amsterdam, v.104, n.1, Oct. 1992.

[LIS 87] LISKOV,B.; GUTTAG, J.V. **Abstraction and Specification in Program Development**. Cambridge: The MIT Press, 1987.

[MAU 89] MAUW, S.; VELTINK, G. J. An Introduction to PSF_d. In: INTERNATIONAL JOINT CONFERENCE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT - TAPSOFT'89, Mar. 1989, Barcelona. **Proceedings....** Berlin: Springer-Verlag, 1989, p.272-285. (Lecture Notes in Computer Science, 352).

[McG 92] MCGREGOR,J.D.; SYKES, D.A. **Object-Oriented Software Development:** Engineering Software for Reuse. New York: Van Nostrand Reinhold, 1992.

[MES 90] MESEGUER, J. A Logical Theory on Concurrent Objects. **SIGPLAN NOTICES**, New York, v.25, n.10, p.101-115, Oct. 1990.

[MES 93] MESEGUER,J.; GOGUEN, J.A. Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation, and Coercion Problems. **Information and Computation**, Orlando, FL, v.103, n.1, p.114-158, Mar. 1993,

[MIL 80] MILNER, R. **Calculus of Communicating Systems**. Berlin: Springer-Verlag, 1980. (Lecture Notes in Computer Science, 92).

[MIL 83] MILNER, R. Calculi for Synchrony and Asynchrony. **Theoretical ComputerScience**, Amsterdam, v.25, n.3, p.267-310, July 1983.

[MIR 91] IRIYALA, K.; HARANDI, M.T. Automatic Derivation of Formal Software Specifications From Informal Descriptions. **IEEE Transactions on Software Engineering**, New York, v.17, n.10, Oct. 1991.

[MOS 88] MOSSES, P. **Unified Algebras and Action Semantics**. Aarhus: Computer Science Dept., Aarhus University, 1988. (Internal Report DAIMI PB-272).

[OLD 86] OLDEROG,E.R.; HOARE, C.A.R. Specification-Oriented Semantics for Communicating Sequential Processes. **Acta Informatica**, Berlin, v.23, n.1, p.9-66, April, 1986.

[PAR 88] PARRINGTON,G.D.; SHRIVASTAVA, S.K. Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems. In: EUROPEAN CONFERENCE ON OBJECT-OERIENTED PROGRAMMING - ECOOP'88, Aug. 1988, Oslo. **Proceedings....** Berlin: Springer-Verlag, 1988. p.233-249. (Lecture Notes in Computer Science, 322).

[PAR 92] PARROW, J.; SJÖDIN, P. Multiway Synchronization Verified with Coupled Simulation. In : INTERNATIONAL CONFERENCE ON CONCURRENCY THEORY - CONCUR'92, 3., Aug. 1992, Stony Brook. **Proceedings....** Berlin: Springer-Verlag, 1992. p.519-533. (Lecture Notes in Computer Science, 630).

[PRE 87] PRESSMAN, R.S. **Software Engineering - A Practitioner's Approach**. New York: McGraw-Hill International, 1987.

[RAM 89] RAMSEY,N. Developing Formally Verified Ada Programs. **ACM SIGSOFT Engineering Notes**, New York, v.14, n.3, May 1989. Trabalho apresentado no INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 5., May 1989, Pittsburgh.

[RIN 92] RINE, D.C.; BHARGAVA, B. Object-Oriented Computing. **Computer**, New York, v.25, n.10, p.6-10, Oct. 1992.

[SAL 92] SALGADO, A.C. et al. Sistemas Hipermidia: Hipertexto e Banco de Dados In: ESCOLA DE COMPUTAÇÃO, 8., 1992, Gramado. **Anais....** Porto Alegre: Instituto de Informática da UFRGS, 1992.

[SAN 92] SANNELLA, D.; SOKOLOWSKI, S. TARLECKI, A. Toward formal development of programs from algebraic specifications: paramete-

risation revisited. **Acta Informatica**, Berlin, v.29, n.8, p.689-736, Dec. 1992.

[SCI 89] SCIORE,E. Object Specilaization. **ACM Transactions on InformationSystems**, New York, v.7, n.2, p.103-122, April 1989.

[SHA 92] SHAPIRO, E. Embeddings Among Concurrent Programming Languages. In: INTERNATIONAL CONFERENCE ON CONCURRENCY THEORY - CONCUR'92, 3., Aug. 1992, Stony Brook. **Proceedings....** Berlin: Springer-Verlag, 1992. p.486-503. (Lecture Notes in Computer Science, 630).

[SHL 88] SHLAER,S.; MELLOR, S.J. **Object-oriented Analysis**: Modeling the World in Data. Englewood Cliffs, NJ: Yourdon Press, 1988.

[SHL 92] SHLAER,S.; MELLOR, S.J. **Object Life Cycles**: Modeling the World in States. Englewood Cliffs, NJ: YourdonPress, 1992.

[SUF 90] SUFRIN, B.; HE, F. Specification, Analysis and Refinement of Interactive Processes. In: HARRISON, M.; THIMBLEBY, H. (Eds.). **Formal Methods in Human-Computer Interaction**. Cambridge: Cambridge University Press, 1990.

[TAK 90] TAKAHASHI, T.; LIESEMBERG, H.K.E. Programação Orientada a Objetos. In: ESCOLA DE COMPUTAÇÃO, 7., 1990, São Paulo. **Anais....** São Paulo: Departamento de Ciência da Computação do IME-USP, 1990.

[VAN 89] VAN GLABBECK, R.; VAANDRAGER, F. Modular Specifications in Process Algebra - with Curious Queues. In: WIRSING, M.; BERGSTRA, J.A. (Eds.). **Algebraic Methods:** Theory, Tools and Applications. Berlin: Springer-Verlag, 1989, p.465-506. (Lecture Notes in Computer Science, 394).

[VAN 89a] VAN HOREBEEK, I.; LEWI, J. **Algebraic Specification in Software Engineering** - An Introduction. Berlin: Springer-Verlag, 1989.

[WAS 89] WASSERMAN, A.I.; PIRCHER, P.A.; MULLER, R.J. An Object-Oriented Structured Design Method for Code Generation. **Software Engineering Notes**, New York, v.14, n.1, p.32-55, Jan. 1989.

[WEC 92] WECHLER, W. **Universal Algebra for Computer Scientists**, Berlin: Springer-Verlag, 1992.

[WEG 88] WEGNER,P.; ZDONIK, S.B. Inheritance as an Incremental Modification Mechanism or WhatLike Is and Isn't Like. In: EUROPEAN CONFERENCE ON OBJECT-OERIENTED PROGRAMMING - ECOOP'88, Aug. 1988, Oslo. **Proceedings....** Berlin: Springer-Verlag, 1988. p.55-77. (Lecture Notes in Computer Science, 322).

[WEG 92] WEGNER, P. Dimensions of Object-Oriented Modeling. **Computer**, New York, v.25, n.10, p.12-20, Oct. 1992.

[WIN 93] WINBLAD, A.L.; EDWARDS, S.D.; KING, D.R. **Software Orientado ao Objeto**. São Paulo: Makron Books, 1993.

[WIN 87] WING, J.M. A Larch Specification of the Library Problem. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 4. **Proceedings....** Los Alamitos, CA: IEEE C. S. Press, 1987. p.34-41.

[WIN 90] WING, J.M. A Specifier's Introduction to Formal Methods. **Computer**, New York, Sept. 1990.

[WIN 88] WINSKEL, G. An introduction to event structures. In: LINEAR TIME, BRANCHING TIME AND PARTIAL ORDER IN LOGICS

AND MODELSFOR CONCURRENCY, May/June 1988, Noordwijkerhout. **Proceedings**... Berlin: Springer-Verlag, 1988. (Lecture Notes in Computer Science, 354).

[WIR 90] WIRFS-BROCK,R.J.; WILKERSON, B.; WIENER, L. **Designing Object-Oriented Software**. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[WIR 90a] WIRFS-BROCK,R.J.; JOHNSON, R.E. Surveying Current Research in Object-Oriented Design. **Comunications of the ACM**, New York, v.33, n.9, p.105-124, Sept. 1990.

[WOL 90] WOLCZKO, M.I. Object-Oriented Languages. In: JONES, C.B.; SHAW, R.C.F. (Eds.). **Case Studies in Systematic Software Development**. London: Prentice-Hall, 1990.

Informática  *UFRGS*

LOP: Uma Abordagem Unificada de Especificação Algébrica, Orientação a Objetos e Processos.

por

Ausberto Silverio Castro Vera

Defesa de Tese apresentada aos Senhores:

Prof. Dr. Edward Hermann Hauesler (PUC/RJ)

Profa. Dra. Laira Vieira Toscani

Prof. Dr. Dalcidio Moraes Claudio

Prof. Dr. Júlio César Ruiz Claeyssen (IM/UFRGS)

Vista e permitida a impressão.
Porto Alegre, 08/12/95.

Prof. Dr. Daltro José Nunes,
Orientador.

Prof José Palazzo Moreira de Oliveira
Coordenador do Curso de Pós Graduação
em Ciência da Computação - CPGCC
Instituto de Informática - UFRGS