APÊNDICE A

A LINGUAGEM CAMILA

A LINGUAGEM CAMILA.

Uma especificação formal escrita na linguagem CAMILA na sua versão mais sim-ples [Almeida e Barbosa 91], que designaremos aqui por CAMILA-0, segue em ge-ral uma estrutura, que consiste na apresentação dos quatro principais compo-nentes da mesma, designadamente:

a) Declaração dos modelos matemáticos das entidades;
b) Definição e identificação do Estado;
c) Definição dos Invariantes;
d) Assinaturas, Pré-condições e Definição das Funções.

No entanto, a linguagem possui também construções que permitem escrever especificações complexas e de "grande escala" de forma modular [Oliveira 91]. Dado que ao longo da tese se usou principalmente o nível não modular, começa-remos por introduzir neste apêndice um resumo da notação usada na definição de cada um dos componentes de uma especificação em CAMILA-0.

A1.- Modelos Matemáticos.

Os modelos matemáticos usados e a correspondente notação CAMILA para a sua definição são apresentados em síntese na tabela seguinte.

Modelo Matemático	**Notação CAMILA**
Conjuntos de X	X-set
Sequências de X	X-list
Funções Finitas de X para Y	X -> Y
Relações Binárias	X <-> Y
Co-Produto (União Disjunta)	T = X | Y | ... ou T = [X]
Produto Cartesiano	T :: X : A Y : B ...
Inteiros	INT
"Strings"	STR
Símbolos	SYM
Universo	ANY

Em geral, quer em CAMILA quer noutras notações associadas a métodos de especificação por modelos, um dos primeiros passos da especificação consiste em associar, através de uma declaração, os identificadores dos tipos dos *objec-tos do problema* aos *modelos matemáticos* que os representam. Em CAMILA-0 tal definição assume a seguinte forma genérica:

Objectos

id_tipo_obj1 = modelo_matemático;

id_tipo_obj2 = modelo_matemático;

....

Assim, e a título de exemplo, é legítima uma declaração da forma,

Objectos

```
Stack =     Elem-list;
Elem =      [Inf];
Inf ::      N : Nome
            C : IdCol -> Colecção
            I : Idade;
Nome =      STR;
IdCol =     SYM;
Colecção =  IdItem-set;
IdItem =    STR;
Idade =     INT;
```

A2.- Identificação do Estado.

A identificação do estado global (?) do sistema sob especificação, é, nesta classe de abordagens, um passo fundamental já que a maior parte das funções que es-pecificam as operações do sistema, excepto as auxiliares, terão funcionalidades que cairão numa de duas classes: *modificadoras* ou *interrogadoras* do estado.

As primeiras deverão aceitar o estado e, possivelmente, outros objectos co-mo parâmetros, e devolver como resultado o estado, eventualmente, alterado.

$$\mathbf{f_m}: ? \ \mathbf{x\ Tp_1\ x\ \ldots\ x\ Tp_n}\ ? \ \ ?$$

As segundas deverão aceitar o estado e, possivelmente, outros objectos co-mo argumentos de entrada e devolver, como resultado, um objecto de dado tipo, sem alteração do estado do sistema, pelo que têm, genericamente a seguinte ari-dade:

$$\mathbf{f_i:}\ ? \ \mathbf{x\ Tp_1\ x\ \ldots\ x\ Tp_n}\ ? \ \ \mathbf{Tr}$$

Assim, a identificação do símbolo que irá representar o estado global (?) na especificação do sistema é feita da forma genérica simples

Estado *id_Estado: Id_Tipo*

sendo de notar que esta forma não pressupõe qualquer declaração, pois não é feita qualquer verificação posterior de coerência, sendo assim apenas uma for-ma de estilo ou um comentário de auxílio à legibilidade.

Usando os modelos matemáticos definidos e operadores associados, e defi-nindo as operações do sistema a especificar como funções matemáticas segun-do uma das formas sintácticas anteriormente apresentadas, então as especifica-ções assim realizadas podem considerar-se de tipo *funcional*.

A3.- Invariantes.

Invariantes são predicados que determinam as condições de validade dos valo-res de entidades de um dado tipo. Para um dado tipo T, o seu invariante, a exis-tir, escreve-se como sendo uma função de nome *inv_T*.

Um invariante de extrema importância é o invariante sobre o *estado,* dado que, por definição, qualquer operação sobre o estado deve partir de um estado válido e devolver como resultado um valor do estado igualmente válido. Uma das primeiras *provas de correcção* a que as definições funcionais das funções modificadoras são submetidas, consiste exactamente na prova de *preservação do invariante*, ou seja, na prova de que o valor do estado resultante da operação satisfaz o invariante do estado. Sendo representados como funções booleanas, a sua estrutura definicional é portanto semelhante à de uma outra qualquer fun-ção, excepção feita à regra do seu identificador, acima referida.

A4.- Funções.

A primeira declaração que é comum realizar-se relativamente a uma função consiste na indicação da sua aridade, ou seja, dos tipos dos parâmetros de en-trada e do tipo do seu resultado, sob a forma genérica,

$$\mathbf{f: T_1 \times T_2 \times .. \times T_n \rightarrow R}$$

De seguida é apresentada a definição da função segundo uma estrutura de-finicional que assume o padrão,

FUNC *id_função* ($\text{id\_par}_1 : \text{Tipo}_1$, ... , $\text{id\_par}_n : \text{Tipo}_n$) : Tipo_Res
PRE *expressão*
RETURNS *expressão;*

Assim, após o cabeçalho pode inscrever-se uma eventual *pré-condição* sob a cláusula PRE, ou seja, um predicado envolvendo tipicamente estado e parâme-tros da função, que, se avaliado como falso, indica que a execução da

função em tal contexto não produz, garantidamente, resultados correctos. A cláusula que se segue, RETURNS, expressa o resultado da função.

As funções de modificação do estado têm a forma geral,

FUNC *id_função* (id_par? : Tipo_? , ... , id_par$_n$: Tipo$_n$) : Tipo_?
PRE *expressão*
STATE *Id_Estado* ? *expressão;*

enquanto que funções de interrogação assumem a forma genérica,

FUNC *id_função* (id_par? : Tipo_? , ... , id_par$_n$: Tipo$_n$) : Tipo$_{n+1}$
PRE *expressão*
RETURNS *expressão;*

Definições contendo ambas as cláusulas STATE e RETURNS, correspondem a definições não funcionais pois implicam a existência de *efeitos laterais* que violam o princípio da transparência referencial[1]. A sua génese é introduzida na secção seguinte.

A5.- Modelos com Estado (Procedimentais).

Ainda que o paradigma funcional de especificação seja sempre claro e de trata-mento matemático simples, a consideração de especificações não-funcionais, en-volvendo a noção explícita de estado é, em muitos casos, de grande utilidade. A linguagem CAMILA, na tradição aliás de linguagens de especificação como VDM ou Z, permite que especificações possam ser construídas usando *modelos com estado*, que deixam de ser álgebras para passarem a ser autómatos de estados finitos (AEF) [Oliveira 93].

O autómato pode passar por um conjunto finito Q de etapas (os *estados*) re-presentáveis por uma tabela de transições de estados,

$$? \; ? \; (Q \times I) \; ? \; (Q \times O)$$

que traduz a evolução do autómato em função da sua reacção à recepção de *es-tímulos* de um conjunto I, reacção que se traduz numa transição para um outro estado no conjunto Q e na eventual produção de um "output" no conjunto O. A possibilidade de a um par (Q x I) se poder associar mais do que um par (Q x O), ou seja, a consideração de indeterminismo no autómato, leva a definir a tabela de transições como uma relação da forma,

[1] Efeitos laterais são de evitar dada a impossibilidade da sua caracterização e tratamento matemático, bem como a bem conhecida degradação das qualidades de legibilidade e compreensão que lhes são associadas.

$$? \; ? \; Q \times I \times Q \times O$$

em vez da função anteriormente apresentada.

O conjunto I (dos *estímulos* ou *eventos*) é um conjunto formado por pares da forma E x A, onde E é um conjunto de identificadores de eventos e A um conjun-to de argumentos possíveis destes. Assim, para cada evento e ? E, teremos que o subconjunto de transições de estado em ? de que este pode ser responsável é dado por

$$?(e) \; ? \; (Q \times A) \times (Q \times O)$$

o que corresponde a afirmar-se que ? se desdobra em tantas relações quantos os *eventos* em E.

Admitindo agora que Q, o conjunto dos estados possíveis do autómato, pode ser representado por um modelo matemático, dado possuir estrutura, cada um dos eventos vai corresponder a uma modificação de tal estrutura, podendo a sua semântica ser estabelecida pela relação

$$? \; ? \; (\text{Estado} \times \text{Argumentos}) \times (\text{Estado} \times \text{Resultado})$$

Eventos (cf. semântica relacional) podem coexistir com funções (cf. semân-tica funcional) numa mesma especificação. Considere-se, por exemplo, a espe-cificação de uma Stack de objectos de tipo X, tendo-se definido como modelo da Stack uma sequência de objectos do tipo X, conforme a declaração

$$\text{Stack} = \text{X-list}$$

Por exemplo para e = POP e Q = Stack, a semântica de POP será dada por

$$?(\text{POP}) \; ? \; \text{Stack} \times (\text{Stack} \times \text{X})$$

ou melhor, dado que POP não está definido em certos estados de Stack, então temos uma relação parcial, ou seja,

$$\text{dom}(?) \; ? \; \text{Estado} \times \text{Argumentos}$$

o que conduz à consideração de um predicado, ou pré-condição, sobre tal even-to, da forma

$$\text{pre-Evento: Estado} \times \text{Argumentos} \; ? \; \text{Bool}$$

Quanto à relação que representa a semântica do evento é usual especificá-la co-mo sendo

pos-Evento: (Estado x Argumentos) x (Estado x Resultado) ? Bool

que é uma relação de *entrada/saída* e não uma função. Passamos assim a ter que considerar na especificação dois estados distintos para a "stack". O estado antes da execução da operação, ? , e o estado após a execução da operação, ?´.

A assinatura do problema passa a poder ser composta por uma sub-assina-tura de tipo funcional (cf. secção *functions*) e por outra procedimental (cf. secção *events*). A *interface com eventos* para a especificação de uma "stack" de elemen-tos de dado tipo é escrita como:

interface STACK ? ELEM
sorts Stack, Bool
functions isEmpty: Stack ? Bool
events INIT : ?
POP : Elem ?
PUSH : ? Elem
end

sendo evidente da definição da sintaxe dos eventos a simplificação realizada pe-la omissão do estado sobre o qual os mesmos actuam.

O modelo com estado (declarado) correspondente, poderia definir-se como:

module STACKLIST (E : ELEM) : STACK
sorts Elem = E.Elem
Stack = Elem-list
State Stack
functions isEmpty(s) Ó s == < >
events INIT Ó ?´= < >
PUSH(e) Ó ?´ = cons(e, ?)
POP(e´) Ó { ? ? < > ? e´ = hd(?) ? ?´ = tl(?)
end

Podendo agora uma operação alterar o estado e calcular um resultado (cf. a operação associada ao evento POP), na linguagem CAMILA estas operações são especificadas recorrendo a definições não-funcionais da forma:

FUNC *id_função* (id_par_1 : Tipo_1 , ..., id_par_n : Tipo_n) : Tipo_Res

PRE *expressão*
STATE Id_Estado ? *expressão*
RETURNS *expressão;*

A6.- Notação Geral.

Apresentam-se de seguida os operadores existentes em CAMILA para a manipu-lação de objectos dos principais modelos matemáticos.

Conjuntos (X-set)

Colecções não ordenadas e sem duplicados de objectos do tipo X, representadas em abstracto como s = { e1, e2, ..., en }.

CAMILA	***Matemática***	***Descrição***
s U r	s ? r	reunião
s * r	s ? r	intersecção
s - r	s - r	diferença
{ }	?	conj. vazio
x in s	x ? s	pertença
x not in s	x ? s	não pertença
subset(s, r)	s ? r	subconjunto
choice(s)		escolha aleatória
the(s)		o único elemento
# s	# s	cardinal
{e1, e2, ..., en }	{e1, e2, ..., en }	enumeração
{ e(x) | x <- s : p(x) }	{ e(x) | x <- s : p(x) }	definição em compreensão
UNION(ss)	s1 ? (s2.. ? sn) ..)	reunião distribuída
s == y	s = y	igualdade
op-orio(e, s)	*op*(e1, ..., *op*(en, e)) .)	redução por *op*

Sequências (X-list)

Colecções s = <s1, s2, ..., sn> ordenadas de objectos do tipo X.

CAMILA	***Matemática***	***Descrição***
hd(s)	s(1)	"head"
tl(s)	<s2, ..., sn>	"tail"
<x:s>		cons(x, s)
<>	<>	seq. vazia
s1 ^ s2		concatenação
elems(s)		elementos de *s*
length(s)		comprimento
plusq(s, f)		alteração
reverse(s)	<sn, ..., s1>	inversão de s
<e1, e2>	<e1, e2>	par
<e1, e2, ..., en >	<e1, e2, ..., en>	enumeração
< e(x) | x <- s ; p(x) >		definição em compreensão
CONC(ss)	s1 ^(s2 ^(.. sn)..)	concatenação distribuída
s == y		
op-orio(e,s)		redução por *op*

Funções Finitas (A ? B)

Correspondências 1:1 e de domínio finito, entre elementos de um conjunto A e elementos de um conjunto B. Podem ser consideradas correspondências defini-das em extensão.

CAMILA	***Matemática***	***Descrição***
dom(f)		domínio
ran(f)	f[dom(f)]	contradomínio
f(x)		aplicação
[]		ff. vazia
f1 == []		ff. vazia?
f\s	[x -> f(x) | x ? (dom(f) - s)]	subtracção ao domínio
f/s	[x -> f(x) | x ? (dom(f) ? s)]	restrição do domínio
f + g		sobreposição
[.. ? ..]		enumeração
[x -> f(x) | x <- A: p(x)]		compreensão

Outros Operadores.

CAMILA	***Matemática***	***Descrição***
add(i, j)	i + j	soma
sub(i, j)	i - j	subtracção
mul(i, j)	i * j	multiplicação
div(i, j)	i / j	divisão

rem(i, j)		resto
abs(i)	| i |	módulo
ascii(s)		código ASCII
atoi(s)		ASCII to INT
itoa(i)		INT to ASCII
x == y	x = y	
x != y	x ? y	
~p	? p	
p ? q, p && q	p ? q	
p V q, p | | q	p ? q	
p => q	p ? q	
all(s <- e : p)	? s ? e. p(s)	
exist(s <- e : p)	?s ? e. p(s)	
exist1(s <- e : p)	$?_1$s ? e. p(s)	

A7.- Notação Modular.

A modularidade para especificação conseguida na linguagem CAMILA versão 1 [Oliveira 91], baseia-se nos conceitos matemáticos de *Interface*, *Módulo* e *Implementação*. Estas construções representam os mecanismos de classificação, de parametrização e de composição necessários a que, a partir da especificação de subsistemas, se possa construir com mais facilidade e estruturação a especi-ficação do sistema desejado.

As *Interfaces* que aqui se consideram são pares I´= (I, ?), em que I é uma in-terface algébrico-funcional, sendo portanto I = (?, X, E), onde ? é uma assinatu-ra algébrica, X uma família indexada de variáveis e E um conjunto de axiomas. A extensão introduzida consiste em considerar-se um conjunto ? de identifica-dores de eventos, disjunto do conjunto S de espécies de ?, e considerar a aplica-ção ? que associa a cada identificador de eventos uma funcionalidade, represen-tada sob a forma S* x S*, ou seja,

? : ? ? S* x S*

De notar no entanto que a funcionalidade de saída de um evento é S*, o que dá a entender claramente o seu carácter procedimental, e não funcional. Para ilustrarmos a utilização das interfaces com eventos em especificações, vamos, por coerência, usar o exemplo da especificação de uma *Queue* (usado no capítu-lo 3 com finalidade idêntica).

Começaremos por apresentar a Interface procedimental para QUEUE, onde se apresentam, as espécies necessárias à especificação, as funções e os eventos, e também um conjunto de axiomas que especificam o TAD QUEUE de forma al-gébrica, ou seja incorporando um conjunto de axiomas sobre os operadores.

```
interface QUEUE ? ELEM
   sorts          Queue, Bool, Nat
   functions      first : Queue ?  Elem
                  len : Queue ?  Nat
                  empty? : Queue ?  Bool
                  inv-Queue : Queue ?  Bool
   events         initq : ?
                  enqueue : Elem ?
                  dequeue : ?  Elem
   axioms
        ? q ? Queue, e ? Elem,
                  first(initq) ? ?
                  dequeue(initq) ? ?
                  len(initq) ? 0
                  empty?(initq) ? true
                  first(enqueue(q, e)) ?  empty?(q)  ? { e          ? [2]
                                       ? empty?(q)  ? { first(q)
                  empty?(enqueue(q, e)) ? false
                  len(enqueue(q, e)) ? 1 + len(q)
                  len(dequeue(q)) ? len(q) - 1
end
```

Como se pode verificar pelo exemplo, eventos associam-se a operações que, tipicamente, trabalham sobre o estado, podendo modificá-lo e também produzir resultados. As funções, quer de consulta do estado, quer auxiliares, quer os in-variantes, têm a sua funcionalidade declarada na parte da Interface correspon-dente à assinatura. Neste caso em que, como vimos anteriormente, podemos ter operações matematicamente "bem comportadas", especificáveis até como fun-ções sobre o estado, faz sentido que se incluam

[2] devem ser tomadas em atenção as notas a propósito de axiomas condicionais introduzidas no capítulo 3.

alguns axiomas, que permitem realizar alguma especificação semântica já ao nível da Interface.

Sendo ELEM uma Interface simples que serve de base à especificação da In-terface QUEUE, consideremos que ela se especifica apenas como,

```
interface ELEM
   sorts          Elem
end
```

dando apenas um nome à espécie que lhe está associada.

A partir das Interfaces devem criar-se então as *Implementações* que passam agora a ser *modelos com eventos*, extensões às ?-álgebras que serviriam de mo-delos caso a especificação fosse algébrico-funcional. *Módulos* são operadores sobre Implementações, permitindo realizar a sua combinação. Implementações básicas, são consideradas constantes ou operadores 0-ários, sendo assim repre-sentáveis como Módulos sem parâmetros.

A estrutura algébrica encontrada é uma meta-assinatura na qual espécies são agora *Interfaces*, constantes *Implementações básicas* ou Módulos constantes e *Módulos* operadores sobre as espécies da assinatura. Para o exemplo em ques-tão, quatro espécies da meta-assinatura devem ser consideradas, em particular, as que correspondem às Interfaces QUEUE e ELEM, a definir, e BOOL e NAT que se assumem pré-existentes, automaticamente incorporáveis em qualquer Interface, definindo uma só espécie, e com uma axiomática bem conhecida. Teremos pois que definir modelos, ou implementações, para as duas Interfaces do problema.

A Interface ELEM aceita um modelo representável pelo módulo constante seguinte,

```
E : ?  ELEM
module E( ) :  ELEM
      sorts          Elem = INT
end
```

O módulo que representa a implementação ou modelo da Interface QUEUE é parametrizado pelo anterior, pelo que a sua assinatura, em correspondência com o que é especificado na respectiva Interface é, Q : ELEM ? QUEUE, defini-do como,

```
module Q (E : ELEM) :  QUEUE
sorts          Elem = E.Elem
```

```
            Queue = Elem-list
State           Queue
functions   first(q) ? { ? empty?(q) ?  hd(q)
            len(q) ? length(q)
            empty?(q) ? q == < >
            inv-Queue(q) ? length(q) < 200
events          initq() ? ?´= < >
            enqueue(e) ? ?´ = ? ^ <e>
            dequeue(e´) ? { ? empty?(q) ?  e´ = hd(?) ? ?´ = tl(?)
end
```

Note-se em particular nas especificações anteriores alguma preocupação no-tacional em distinguir entre argumentos de entrada e de saída, e valor do estado antes e depois da execução da operação. Estado após a operação representa-se pela variável ? decorada, ou seja, como ?´. Os identificadores de argumentos não decorados são argumentos de entrada, sendo os decorados argumentos de saída tal como na operação *dequeue*.

De notar ainda na especificação de *dequeue* que uma conectiva lógica indi-ca que um resultado é calculado e colocado no argumento de saída, sendo o es-tado alterado da forma definida. Finalmente, note-se que o que se acaba de es-pecificar é um Módulo parametrizado, pelo que as espécies da sua Interface de-vem ser associadas às espécies dos Módulos parâmetro, o que é realizado usan-do a cláusula *sorts*. Note-se ainda a indicação expressa do identificador do *esta-do,* no exemplo Queue.

Repare-se ainda que os termos da meta-assinatura representam especifica-ções que combinam outras subespecificações. Por exemplo, Q(E) designa a espe-cificação, com interface QUEUE, de uma "queue" de inteiros, dado que o módu-lo E se associa, no exemplo, a tal tipo de dados.