APÊNDICE C

ESPECIFICAÇÃO DO AGSYS

Sistema Gráfico baseado em Arquétipos

C.1.- AGSYS: Apresentação.

O sistema gráfico simbólico designado por AGSYS tem o seu funcionamento baseado no conceito de arquétipo e no paradigma de interacção por edição estrutural que se apresentou no capítulo 5 desta tese.

Para além desta filosofia básica, várias foram as extensões funcionais que tiveram que ser realizadas por forma a tornar uma simples, ainda que promisso-ra, ideia num sistema realmente operacional. As principais extensões introduzi-das são aqui referidas informalmente, podendo no entanto verificar-se como constando da funcionalidade final do sistema cuja especificação formal se apre-senta na secção seguinte.

1) Base de Dados de arquétipos;

2) Arquétipos como operadores;

3) Instanciação por defeito;

4) Múltiplas representações internas.

C.2.- AGSYS: Especificação Formal.

A especificação formal do AGSYS, é aqui apresentada usando, para a especifica-ção das funções a linguagem CAMILA [Almeida e Barbosa 91] com algumas ex-tensões que visam facilitar a construção e apresentação modular da especifica-ção.

A linguagem CAMILA tem implementados mecanismos de modularidade, tal como se apresentou no APÊNDICE A. No entanto, adopta-se aqui um esquema de modularidade mais simples e convencional, com o objectivo de facilitar a com-preensão da especificação apresentada.

Assim, cada *módulo de especificação*, MSPEC, a utilizar na especificação, vai definir o conjunto de objectos e funções que são importadas de outros módulos, segundo a estrutura apresentada a seguir, onde se distingue de forma clara o que é importado pelo módulo e o que neste é definido.

MSPEC: < *nome de módulo* >

IMPORTA

Objectos < *lista de funções* > ***de*** < *nome de módulo* >
Opers < *lista de funções* > ***de*** < *nome de módulo* >

DEFINE

Estado: < *identificador do estado* >
Objectos < *lista de pares identificador de objecto, modelo matemático* >
Invariantes < *invariantes a observar* >
Operações <*definições formais das operações*>

FIM MSPEC: < *nome de módulo* > ;

A cláusula de importação, *IMPORTA*, estabelece entre os módulos de especi-ficação uma relação de inclusão associada aos objectos e operações nela decla-rados e, consequentemente, uma relação estrutural hierárquica não recursiva.

A especificação será apresentada de forma bottom-up, ou seja, apresentan-do primeiro os módulos-folha e de seguida os módulos-raiz. A hierarquia dos principais módulos é apresentada na figura C.1 para que a especificação possa ser melhor seguida. Módulos eventualmente referidos mas não apresentados, como por exemplo REL, LIST, MAT, STRING, SEQ, etc., correspondem a módulos pré-definidos da linguagem de especificação ou módulos considerados básicos.

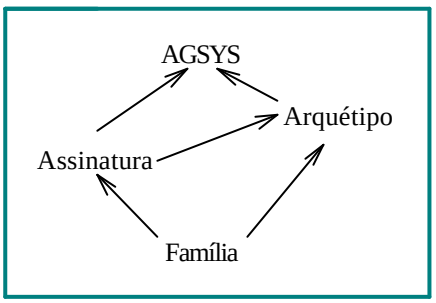

Fig. C.1 - Hierarquia da Especificação.

Note-se que a relacção hierárquica apresentada na figura C.1 corresponde a uma relação *importa-de* qualitativa, ou seja, segundo a qual objectos e opera-ções podem ser importados por designação nominal (ou seja por selecção).

Teremos assim a especificação:

MSPEC: *Família (Indexada)*
IMPORTA
Opers
projd ***de*** Rel

DEFINE

; ***Estado:*** dfm(A, B) ; *família disjunta e não vazia de A para B*

Objectos
fam(A, B) = A -> B-set
dfm(A, B) = { f | f ? fam(A, B) : é-disjunta(f) ? não-vazia(f) }

FUNC é-disjunta(f: fam(A, B)) : Bool
RETURNS all(a1 <- dom(f); a2 <- dom(f) - {a1} : f[a1] * f[a2] == {});

FUNC não-vazia(f: fam(A, B)) : Bool
RETURNS all(a <- dom(f) : ~ (f[a] == {}));

Operações

FUNC allElem(f: fam(A, B)) : B-set
; *conjunto dos elementos do contradomínio*
RETURNS UNION({ projr({a}, f) | a <- dom(f)});

FUNC index(b: B, f: fam(A, B)) : A
; *determina o índice de um dado elemento de B*
RETURNS the({ a | a <- dom(f) : b ? f[a] });

FUNC addElem(a: A, b: B, f: fam(A, B)) : fam(A, B)
; *junta um elemento b de índice a*
PRE all(a1 <- dom(f) : a != a1 => f[a1] * {b} == {})
RETURNS f + [a -> $f[a]_{\{\}}$ U {b}];

FUNC mkFamfRel(rel : A <-> B) : fam(A, B)
; *transforma uma relação binária numa família*
RETURNS [a -> projd({a}, rel) | a <- dom(rel)];

FUNC U_f(f1: fam(A, B), f2: fam(A, B)) : fam(A, B)
; *reunião de famílias indexadas*
RETURNS [x -> $f1[x]_{\{\}}$ U $f2[x]_{\{\}}$ | x ? (dom(f1) U dom(f2))];

FUNC $UNION_f$(sf: fam(A, B)-set) : fam(A, B)
; *reunião distribuída de famílias*
RETURNS U_f-orio({}, sf);

FIM MSPEC: Família

MSPEC: *Assinatura*

IMPORTA

Objectos
dfm(A, B) ***de*** Família
Opers
index, allElem ***de*** Família

DEFINE

; ***Estado:*** ?ig

Objectos
?ig = dfm(Fun, Op-set); *assinatura como família indexada*
Sort = SYM;
Op = SYM;
Fun :: I: Sort-list O: Sort; *espécies de entrada e de saída*

Invariantes

FUNC inv-?ig(sig: ?ig) : Bool
; *numa assinatura deve existir pelo menos um operador por cada funcionalidade.*
; *o conjunto de chegada de todas as operações coincide com o conjunto das espécies.*
RETURNS ~ exist(f <- dom(sig) : sig(f) == {}) ? (rs(sig) == allSorts(sig)

Operações

FUNC enrich(f: Fun, s: Op-set, sig: ?ig) : ?ig
; *junta um conjunto de operadores de funcionalidade f*
RETURNS sig + [f -> $(sig[f]_{\{\}}$ U s)];

FUNC rs(sig: ?ig) : Sort-set
; *dá como resultado todas as espécies resultado dos operadores da assinatura*
RETURNS { O(f) | f <- dom(sig) };

FUNC allSorts(sig: ?ig) : Sort-set
; *dá como resultado todas as espécies envolvidas nos operadores da assinatura*
RETURNS rs(sig) U UNION({ elems(I(f)) | f <- dom(sig) });

FUNC allOps(sig: ?ig) : Op-set
; *dá como resultado todos os operadores da assinatura*
RETURNS allElem(sig);

FUNC opsR(s: Sort, sig: ?ig) : Op-set
; *dá como resultado todos os operadores que têm a espécie s como resultado*

```
RETURNS UNION( { sig[f] | f <- dom(sig) : s == O(f) } );



FUNC opsL(sl: Sort-list, sig: ?ig) : Op-set
; dá como resultado todos os operadores que têm as espécies da lista sl
; como funcionalidade de entrada
RETURNS UNION( { sig[f] | f <- dom(sig) : sl == I(f) } );

FUNC nullOps(sig: ?ig) : Op-set
; dá como resultado todos os operadores 0-ários ou constantes, ou seja, que têm
; funcionalidade de entrada nula
RETURNS opsL(<>, sig);

FUNC func(op: Op, sig: ?ig) : Fun
; determina a funcionalidade de um operador
PRE op ? allOps(sig)
RETURNS index(op, sig);

FUNC oSort(op: Op, sig: ?ig) : Sort
; determina a espécie resultado de um operador
PRE op ? allOps(sig)
RETURNS O(func(op, sig));

FUNC argSorts(op: Op, sig: ?ig) : Sort-list
; determina as espécies dos argumentos de um operador
PRE op ? allOps(sig)
RETURNS I(func(op, sig));

FUNC opsbyfunc(sl: Sort-list, s: Sort, sig: ?ig) : Op-set
; operadores com a funcionalidade dada
RETURNS sig[Func(sl,s)]{};
```

FIM MSPEC: Assinatura ;

MSPEC: *Arquétipo*

IMPORTA

Objectos

dfm(A, B) ***de*** Família
?ig, Sort, Op ***de*** Assinatura

Opers

allOps, argSorts, oSort ***de*** Assinatura
index, allElem, mkFamfRel, é-disjunta ***de*** Família
mkRelfpl ***de*** Rel

DEFINE

; ***Estado:*** Arq

Objectos

```
Arq = W?x ; um arquétipo é um termo com variáveis
W?x = X | ?t
?t :: O: Op A: W?x-list ; recursividade => árvore
X = SYM;
Var = dfm(Sort, X);
W? :: O: Op A: W?-list ; termos de base, i.é., sem variáveis
```

Facto

```
W? ? W?x
```

Invariantes

```
FUNC inv-Arq(arq: W?x, sig: ?ig, v: Var) : Bool
; os operadores de um arquétipo devem pertencer ao conjunto dos operadores da
; assinatura; as variáveis devem pertencer ao conjunto de variáveis Var.
; as espécies dos argumentos de um dado operador deve coincidir com as espécies
; resultado de cada uma das subárvores do operador.
RETURNS
        (treeOps(arq) ? allOps(sig)) ? treeVars(arq) ? allVars(v) ?
        if is-X(arq)  then  true
                      else  let (op = O(arq), ls = A(arq))
                            in
                            argSorts(op, sig) == < treeSort(x, sig, v) | x <- ls > ?
                            all(t <- elems(ls) : inv-Arq(t, sig, v) ?
                                                 inv1-Arq(arq, sig));

FUNC inv1-Arq(arq: W?x, sig: ?ig) : Bool
```

```
; o arquétipo é bem formado
RETURNS
        aridadesOk(arq, sig) ? tiposOk(arq, sig) ? varsOk(arq, sig);
```

sendo

```
FUNC aridadesOk(arq: W?x, sig: ?ig) : Bool
PRE treeOps(arq) ? allOps(sig)
RETURNS if is-X(arq)   then true
                       else   length(argSorts(O(arq), sig)) == length(A(arq)) ?
                              all( t <- A(arq) : aridadesOk(t, sig));

FUNC tiposOk(arq: W?x, sig: ?ig) : Bool
RETURNS if is-X(arq)   then true
                       else   let (rel = mkRelfpl(argSorts(O(arq), sig), A(arq)))
                              in all( (s, tt) <- rel : is-?t(tt) => s == oSort(O(tt), sig) ?
                                 typesOk(tt, sig) );

FUNC varsOk(arq: W?x, sig: ?ig) : Bool
PRE (treeOps(arq) ? allOps(sig)) ? aridadesOk(arq, sig)
RETURNS é-disjunta(yctxt(arq, sig));
```

Operações

```
FUNC allVar(v: Var) : X-set
; todas as variáveis da família v
RETURNS allElem(v);

FUNC varSort(x: SYM, v: Var) : Sort
; determina a espécie de dada variável
PRE x ? allVar(v)
RETURNS index(x, v);

FUNC treeVars(arq: W?x) : X-set
; determina o conjunto das variáveis do arquétipo
RETURNS
      if is-X(arq)   then {arq}
                     else UNION ({ elems(< treeVars(t) | t <- A(arq) >) });

FUNC treeOps(arq: W?x) : Op-set
; determina o conjunto de operadores do arquétipo
RETURNS
      if is-X(arq)   then {}
                     else {O(arq)} U UNION( { elems(<treeOps(t) | t <- A(arq)>)} );

FUNC treeSort(arq: W?x, sig: ?ig, v: Var) : Sort
; determina a espécie do arquétipo, ou seja, a espécie resultado do operador que
; é a raiz da árvore, ou, caso seja apenas uma variável, dessa variável
PRE if is-X(arq) then arq ? allVar(v) else O(arq) ? allOps(sig)
RETURNS  if is-X(arq)   then varSort(arq, v)
```

```
                    else oSort(O(arq), sig);

FUNC yctxt(t: ?t, sig: ?ig) : Var
; esta função implementa o mecanismo de inferência das espécies das variáveis
; contidas num arquétipo, cuja árvore tem na raiz um operador, construindo assim
; o seu contexto
PRE treeOps(t) ? allOps(sig) ? aritiesOk(t, sig)
RETURNS let ( rel = mkRelfpl(argSorts(O(t), sig), A(t)),
              rvars = { (first(p), second(p)) | p <- rel : is-X(second(p)) },
              tms = ran(rel - rvars))
        in mkFamfRel(rx) U_f UNION_f({ yctxt(tm, sig) | tm <- tms});

FUNC inst(arq: W?x, bind: X -> W?x) : W?x
; dado uma função finita que representa um conjunto de associações de termos a
; variáveis, o arquétipo parâmetro é instanciado, substituindo-se cada variável em X
; pelo termo que lhe está associado em bind.
RETURNS if is-X(arq) then bind[arq]_arq
                     else ?t(O(arq), < inst(tt, bind) | tt <- A(arq) >);
```

FIM MSPEC: Arquétipo ;

MSPEC: *AGSYS*

IMPORTA

Objectos

W?x, X, Var ***de*** Arquétipo
?ig, Sort, Op ***de*** Assinatura

Opers

opsRnull, argSorts, opsR, opsL ***de*** Assinatura
allVar, inst, treeSort, treeVars, treeOps ***de*** Arquétipo
mkFamfRel, é-disjunta ***de*** Família
mkRelfpl ***de*** Rel

DEFINE

; ***Estado:*** Agsys

Objectos

Agsys :: G: Gdb		*Base de Dados do Sistema Gráfico*
N: [Nucl]		*Ambiente de trabalho não persistente*
B: [Buff] ;		*Dados correntemente em manipulação*
Gdb =	Envid -> Env ;	*Cada contexto de trabalho é identificado*
Nucl ::	I: Envid	*Identificador do ambiente de trabalho*
	E: Env ;	*Ambiente actual de trabalho*
Env ::	S: ?ig	*Assinatura*
	M: ??	*Função significado dos termos*
	D: ?A	*Operadores derivados*
	K: Knlg ;	*Directoria de termos*
Anm =	SYM ;	*Identificador de arquétipo*
?Arch ::	L: X-list	*Representação de arquétipos como sendo*
	R: d?t ;	*expressões ?*
dW?x =	X | d?t ;	*Operadores derivados*
d?t ::	O: (Op | Anm)	

```
            A: dW?x-list;

Knlg ::     D: W?dir
            I: Anm -> Inst;
Inst =      X -> W?

Buff ::     N: [X]              Buffer para manipulação de termos
            C: Path             Cursor sobre o termo
            T: [dW?x]           Termo em manipulação
Path =      Nat-list ;          Sequência dos ramos até ao cursor


W?dir = X -> dW?x ;             Directoria de termos identificados

W? ::       O: Op               Termos base da assinatura
            A: W?;

Var =       dfm(Sort, X) ;      Família indexada pelas espécies
Envid =     SYM;
?A =    Anm -> ?Arch ;          Identificadores de arquétipos e sua
                     ;          representação como ? termos

dW?x1 =     X | d?t1 ;          Arquétipo em formato buffer
d?t1 ::     O: (Op | Anm)
            F: Nat -> dW?x1;

W?x := X | W?t ;                Termo com variáveis
W?t ::      O: Op
            A: W?x;
```

Invariantes

```
FUNC inv-W?dir(d: W?dir) : Bool
; não pode existir recursividade na directoria (versão não transitiva)
RETURNS norecursion(d);

FUNC norecursion(d: W?dir) : Bool
; toda a variável a que se associa um termo não faz parte do conjunto de
; variáveis do próprio termo
RETURNS all(x <- dom(d) : x ?  treeVars(d[x]));

FUNC inv-?A(d: ?A) : Bool
; não pode existir recursividade na directoria de arquétipos (versão não transitiva)
RETURNS norecursionA(d);

FUNC norecursionA(d: ?A) : Bool
; nenhum identificador de arquétipo faz parte dos identificadores
; de arquétipos contidos no termo
RETURNS all(aid <- dom(d) : aid ?  treeArchs(R(d[x])))
FUNC treeArchs(buf: Buff) : Bool
; o estado do cursor é uma sequência válida de ramos
RETURNS C(buf) ?  allpaths(T(b));

FUNC treeArchs(arq: dW?x) : Anm-set
; dá o conjunto de nomes de arquétipos do operador derivado
```

```
RETURNS { op | op <- treeOps(arq) ; is-Anm(op) };

FUNC inv-Buff(buf: Buff) : Bool
; o estado do cursor é uma sequência válida de ramos
RETURNS C(buf) ?  allpaths(T(b));
```

Operações

```
FUNC buildobj(sys: Agsys, x: X) : Agsys
; constrói um objecto

RETURNS
        let     (   env = E(N(sys)), sig = S(env), term = sdedit(sig),
                    newbuf = Buff(x, <>, ldbuf(term)) )
        in
            Agsys(G(sys), N(sys), newbuf);

FUNC sdedit(sig: ?ig) : W?      ; edição dirigida pela estrutura
; constrói um termo base da assinatura
RETURNS
        let ( sort = inpFrom(rs(sig) ) ; lê a espécie do termo a construir
        in
            ibuildgrnt(sort, sig) ; constrói o termo base interactivamente

FUNC inpFrom(s: Y-set) : Y
; lê um valor obrigatoriamente no conjunto s
RETURNS
        ppickone(s, "escolha 1 de :");

FUNC ibuildgrnt(s: Sort, sig: ?ig) : W?
; interactivamente constrói um termo base da assinatura
RETURNS
        let ( op = inpFrom(opsR(s, sig)) )
        in
            W?(op, < ibuildgrnt(esp, sig) | esp <- argSorts(op, sig) >);

FUNC ibuildarq(s: Sort, sig: ?ig, vars: Var) : W?x
; interactivamente constrói um arquétipo
RETURNS
        let ( tipo = inpFrom({Var, Termo}) )
        in
        if tipo == Var
        then inpFrom(vars[s])
        else
            let ( op = inpFrom(opsR(s, sig)) )
            in
                W?x(op, < ibuildarq(esp, sig) | esp <- argSorts(op, sig) >);

FUNC ldbuf(t: W?x) : W?x1
; converte termo com variáveis para formato buffer
```

```
RETURNS
            if is-X(t)
                then t
                else W?x1(O(t), ff_from_list( <ldbuf(st) | st <- A(t)> ));

FUNC root(buf: Buff) : Buff
; posiciona o cursor na raiz do termo => igual a NIL
RETURNS
            Buff(N(buf), <>, T(buf));



FUNC up(buf: Buff) : Buff
; posiciona o cursor no nodo superior
RETURNS
            let ( cursor = C(buf) )
            in
                if cursor == <> then buf
                                else Buff(N(buf), blast(cursor), T(buf));

FUNC down(buf: Buff) : Buff
; posiciona o cursor na raiz da subárvore mais à esquerda do nodo
RETURNS
            let ( cursor = C(buf), term = tfpath(cursor, T(buf)) )
            in
                if is-X(term)
                    then buf
                    else
                        if F(term) == []
                            then buf
                            else Buff(N(buf), cursor ? <1>, T(buf));

FUNC left(buf: Buff) : Buff
; posiciona o cursor na subárvore à esquerda do nodo
RETURNS
            let ( cursor = C(buf) )
            in
                if (cursor == <>) ? (last(cursor) == 1)
                    then buf
                    else
                        let ( newlast = last(cursor) - 1 )
                        in
                            Buff(N(buf), blast(cursor) ? <newlast> , T(buf));

FUNC right(buf: Buff) : Buff
; posiciona o cursor na subárvore à direita do nodo
RETURNS
            let ( cursor = C(buf) )
            in
                if (cursor == <>)
                    then buf
                    else
                        let (   newlast = last(cursor) + 1,
                                t = tfpath(blast(cursor), T(buf) )
                        in
                            if ~(newlast ? dom(F(t)))
```

```
                        then buf
                        else Buff(N(buf), blast(cursor) ? <newlast> , T(buf));

FUNC inst(t: W?x, inst: X ? W?) : W?
; dado um mapa de variáveis e seus valores, Inst, esta função instancia
; completamente um arquétipo.
RETURNS
        if is-X(t)
            then  inst[t]t
            else  W?(O(t), <inst(tt, Inst) | tt ? A(t)>);

FUNC finst(sys: Agsys, arq: Anm, x: X) : Agsys
RETURNS
        let (  env = E(N(sys)), buf = B(sys), Dop = D(env), sig = S(env),
               tdir = D(K(env)), vars = L(Dop[arq]),
               sorts = argSortsA(arq, sig, Dop),
               consts = < userChoice(opsRnull(s, sig)) | s ? sorts >
               assocs = ff_from_lists(vars, consts)
               prot = inst(R(Dop[arq], assocs)
               newbuf = Buff(x, <>, ldbuf(prot))
               newtdir = tdir + [x -> prot]
               nidir = I(K(env)) + [arq -> [Dop[arq] + [x -> consts]]]
               newenv = Env(sig, M(env), Dop, Knlg(newtdir, nidir))
        in
               Agsys(G(sys), Nucl(I(N(sys)), newenv), newbuf);

FUNC mk_context(t: d?t, sig: ?ig, dop: ?A) : dfm(Sort, X)
RETURNS
        let (  ? = O(t),
               ss = if is-Op(?) then argSorts(?, sig)
                                else argSortsA(?, sig, dop),
               r = mkRelfpl(ss, A(t))
               rx = { (s, st) | (s, st) <- r ; is-X(st) }
               ts = ran(r-rx)
        in mkFamfRel(rx) ?f UNIONf({mk_context(tt, sig, dop) | tt <- ts});

FUNC heval(t: W?, ?: ??, sig: ?ig) : SemExp
RETURNS
        let (  ? = O(t),
               opSort = oSort(?, sig)
               homo = ?[opSort]
               semExp = homo[?]
        in
            if A(t) == <>   then semExp
                            else <semExp> ? <heval(tt, ?, sig) | tt <- A(t)>;

FUNC init( ) : Agsys
RETURNS Agsys( [], nil, nil );
```

MSPEC: *AGSYS-RunTime*

IMPORTA

Objectos
AGSYS
Opers
AGSYS

DEFINE

Estado: agsys: Agsys

Operações

FUNC !init( ) : SYM
STATE agsys ? Agsys([], nil, nil);

FUNC !finst( ) : SYM
STATE agsys ? finst(agsys, arq, x);

FUNC !buildobj( ) : SYM
STATE agsys ? buildobj(agsys, arq, x);

FUNC !root( ) : SYM
STATE agsys ? Agsys(G(agsys), N(agsys), root(B(agsys));

FUNC !up( ) : SYM
STATE agsys ? Agsys(G(agsys), N(agsys), up(B(agsys));

FUNC !down( ) : SYM
STATE agsys ? Agsys(G(agsys), N(agsys), down(B(agsys));

FUNC !left( ) : SYM
STATE agsys ? Agsys(G(agsys), N(agsys), left(B(agsys));

FUNC !right( ) : SYM

STATE agsys ? Agsys(G(agsys), N(agsys), right(B(agsys));