Algebraic Specification of Documents

José Carlos Ramalho jcr@di.uminho.pt
José João Almeida jj@di.uminho.pt
Pedro Henriques prh@di.uminho.pt

Departamento de Informática
Universidade do Minho
Braga – Portugal
October 4, 1996

Abstract

It is becoming normal that a document should serve several purposes. However, the majority of available text processors is purpose-oriented, reducing the necessary flexibility and reusability of documents. Some waste of time arises from adapting the same text to each different purpose, when this task could be done automatically (from the first version of the document) with an appropriate system.

This communication highlights the guidelines to build a system to solve the above problem. Such a system should be an *algebraic based environment* and provide facilities for:

- Document type definitions;
- Definition of Function over document types;
- Document definitions as algebraic terms.

This approach (*rooted in the tradition of constructive algebraic specification models*), will allow for an homogeneous environment to deal with operations such as *merging* documents, *converting* formats, *translating* documents, *extracting different kinds of information* (to set up information repositories, data bases, or semantic networks) or *portions of documents* (as it happens, for instance, in *literate programming*), and some other actions, not so traditional like *mail reply*, or *memo production*.

We intend to use Camila (a specification language and prototyping environment developed at Universidade do Minho, by the Computer Science group) to develop the above mentioned system.

1 Introduction

A **document** is a collection of pieces of text —a pure character string— organized according to a specific structure. It contains a message to be delivered (to someone), and its structure is defined in order to enhance some special parts of that message, and in general, to improve its transmission process.

Document processing means transforming a given document in order to produce another document (with a different structure or with the same organization expressed in a different markup language) or to execute some reactive action. This definition includes tasks like *text formatting*, *translation*, *interpretation*, *automatic reply to message*, *literate programming*, and so on. Therefore a document processor is no more than a typical language processor.

Algebraic programming is an approach to (computer) problem solving based on the definition of an algebraic model to specify the entities and transformations arising from the problem being considered.

In this context, a model consists of an algebra(Goguen, Thatcher, and Wagner, 1978) (or a relational structure(Nipkow, 1986)) for a given signature (i.e. a set S of sorts and an S^*-indexed set of operation symbols). The model assigns consistently a set to each sort symbol and a function (or a relation) to each operation symbol.

Our intuition suggests that a document can be thought of as a data element and document processing as an algebraic operation.

Therefore, we propose to apply the algebraic specification method to document processing. The key idea of this approach is the definition of a *document type* —every document must have an associated type, predefined or userdefined. Each processing task is specified as an operator (a function) defined over document types, and a document can be expressed as a term of the underlying algebra. This system will be at least useful for specifying documents and tools and rapidly prototype them.

Furthermore, we will also analyze the use of an external standard format to describe documents. We will propose a mapping between this external document markup system and the internal algebraic typing system. As we already know how to refine algebraic functions into procedural programming languages, that mapping will enable to formally reach implementations from specifications and prototypes.

These concepts —document types, functions and documents— are discussed in detail in the next section (sec. 2). The architecture of the algebraic system we envisage to develop and its interface to the real world of document manipulation is described in section 3. In section 4 we illustrate our proposal with two examples. The paper is concluded with some final remarks and prospect for future work (sec. 5).

2 The proposed algebraic approach

Document definition is an old problem.

Whoever uses a computer to carry out the tasks involved in document production wants **easy manipulation of documents** (subdocument extraction, structural document translation).

Furthermore, the tools used to manipulate documents should be correct and reusable.

Document correctness is difficult to define. We have to guarantee:

- structural correctness – have the right components according to text purpose. It is important to have an explicit structure visible in the document (in order to allow for automatic structure recognition) [1].
- invariant preservation – defined constraints to be satisfied by the document.

Document reuse arises when one has to deal with different documents based on the same text, or different views.

To achieve this it is necessary to separate a document from the details of final views.

[1]This is the main goal of markup languages

Ex. a dictionary can be printed. However, its definition should not be tied to pagination, because that would disable the possibility of using it for other operations as an electronic hyper-text.

2.1 Document type definitions

The document type definition is a way to define properties to be verified by document instances. Therefore it is a step to the notion of correctness.

In Camila (Barbosa and Almeida, 1995) (a model-based algebraic specification, briefly introduced in appendix A) a type definition involves the definition of:

- the carrier set of its sort
- an invariant (a boolean-valued function that restricts the carrier set cardinality to cope with semantic requirements).

Ex: 1 [**electronic mail**]
When dealing with electronic mail system specification, we must define the `mail` sort. The carrier set definition is:

```
mail= header : id -> string        (1)
      body   : string-seq          (2)
```

`(1)` - mapping between id(entifier) and strings.
`(2)` - a list of strings (lines).

The following invariants guarantee that messages have no null content:

```
inv_mail(m) = body(m) != "" \/
              header(m)[subject] != "" ;
```

A complete example can be found in section 4.1.

———————

In order to be consistent with this model, a document has to be structurally correct, and guarantee the invariant.

The structure of a document is also a good guide to build translations to/from other formats/models, manipulation functions and browsers of documents.

2.2 Function definition over document types

The specification of new functions over document types is a way to:

- describe document manipulation (as translation between different formats)

- describe the behavior (or intended behavior) of existing tools

- discuss future tools and (document) types

- build documentation of tools and formats

In our framework, Camila , the definition of a function comprises the following steps:

- definition of its signature

- definition of a precondition (predicate over input parameter and state) that has to be evaluated to `true`, so that the function can be applied

- definition of a returned value

- definition of how to update of the state variable

Function specifications are an important step in the definition of system correctness, because, in order to be correct, a function must:

- have the right signature

- verify its precondition (the predicate must be true)

- guarantee that the invariant of the returned value type evaluates to `true`

- guarantee that the invariant of the computed state variable evaluates to `true`.

- guarantee that the precondition of every function used is true

The basic collections of operators associated to Camila type constructors (eg, union, intersection of two sets, domain or range of binary relations, application or overwrite of finite functions, etc) are available as primitive functions in the language. So are the propositional connectives and quantifiers.

The availability of all the repertoire of Camila operators and the guidelines offered by the type model greatly simplifies the task of defining a new function.

3 The Algebraic System

It is quite clear that an algebraic system is a little far from reality, concerning document electronic interchange. This entails the need of one more layer intended to establish the bridge between the algebraic system and the outside world of documents. A format (or set of formats) must be chosen as the input and output of this layer and consequently of the system. This format should not have any character set dependencies and should be easy to parse and generate. This layer will incorporate a parser/translator for the chosen formats.

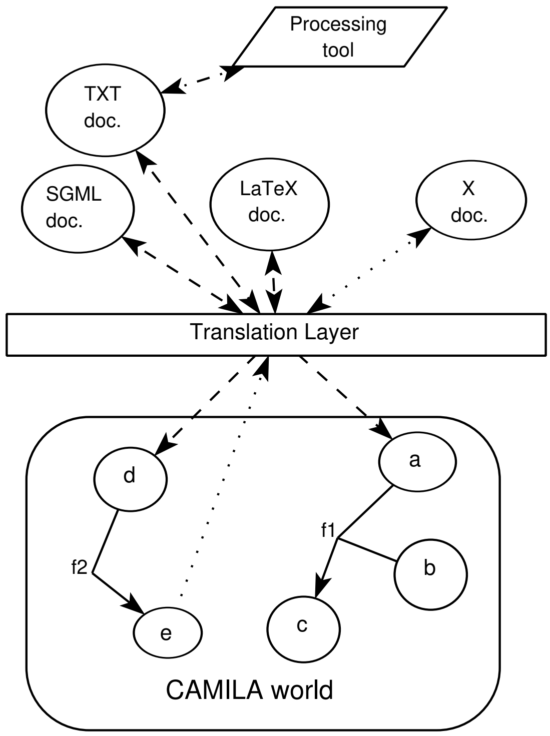

Figure 1: System Architecture

Fig. 1, gives a more detailed idea of the intended system.

f1 stands for a Camila function that receives two documents as arguments and produce a new one.

f2 Camila function that transform one document in another.

External processing using external tools (accepting a format FMT1 and building a document in format FMT2), is done by writing an exportFMT1 and a importFMT2 functions (see txtedit in mail example).

Looking at the actual scene there are some strong candidates like LaTeX(Lamport, 1986), Word or SGML (Sperberg-McQueen and Burnard, 1994), to be considered as an input/output format to/from our system. On the other hand, a closer look to those formats shows that Word is not a good choice because it has not a visible structure and its format (Microsoft copyright) is not well known to the public.

Both SGML and LaTeX have a visible structure, are widely used, and there are plenty of tools able to process documents written in their format.

Comparing this two one major difference comes up. LaTeX is too much tied up to format and typographic aspects, and SGML is not. Besides that SGML has the following advantages:

- it is an ISO standard (ISO 8879).
- it is not concerned with formatting aspects and is fully data independent.
- its only concern is the textual structure of a document.
- its use is spreading rapidly, and there are many commercial and public-domain tools RITA(Cowan et al., 1991), CoST(Harbo, 1994) and sgmlpl(Megginson, 1995b; Megginson, 1995a) available to create and process SGML documents.

Therefore, for the time being, SGML is the base format chosen to communicate with the outside world (this will not eliminate the possibility of adding other formats).

3.1 SGML as Input and Output

SGML, abbreviation for "Standard Markup Language", is a meta-language to define descriptive markup languages which specifies the structure of a particular kind of document. The markup language does not specify how the document is to be processed or printed, it only specifies its structural elements and the relations between them. For example, a markup language could specify the lines and stanzas of a poem, but not the type of font or size to be used when printing or displaying the document.

Using SGML it is possible to specify the structure of a certain kind of document creating a Document Type Definition (DTD). Documents that obey that structure are classified as being of that type. This way, any SGML document belongs to a class/type of documents.

Therefore, we can say that a DTD corresponds to a signature of an algebraic specification.

When creating a DTD, structural elements and the way they are related are specified. Then when writing a document according to a specific DTD, we decorate it with start tags ("<tag>") and end tags ("</tag>") delimiting the structural elements, as, for example, the following mail message:

Ex: 2 [**electronic mail**]

```
<mail>
  <header>
    <from> jcr@di.uminho.pt </from>
    <to> epl@di.uminho.pt </to>
  </header>
  <body>
    This is only a tutorial example
    to be used in this article...
  </body>
</mail>
```

———

The correspondent DTD may be specified as:

Ex: 3 [**Mail DTD**]

```
  <!ELEMENT mail   - - (header,body)>
/* mail is composed by header and body */
  <!ELEMENT header - - (from,to)>
/* header is composed by from and to   */
  <!ELEMENT from   - - (#PCDATA)>
/* #PCDATA is free text                */
  <!ELEMENT to     - - (#PCDATA)>
  <!ELEMENT body   - - (#PCDATA)>
```

———

Assuming SGML as the standard format for input/output to/from the system, we need to establish a correspondence function between SGML elements and the Abstract Data Types of the Algebraic System. To do this, we must ensure that SGML specification language is self-contained in Camila data models, in the next section, we will establish a correspondence function between SGML and CAMILA Data Models.

3.2 SGML $\leftrightarrow$ CAMILA DATA MODELS

SGML is a very simple language, structure oriented. So it should not be difficult to create a correspondence between each one of its features and CAMILA data models.

An SGML specification is composed by a series of ELEMENT declarations. Each ELEMENT corresponds to a structural element of the document and is defined as text or as being a combination of other elements. SGML as a few operators to specify relations between elements:

SGML features	
Expression	Meaning
x, y	element x followed by element y
x & y	x and y in any order
$x \mid y$	either x or y
$x*$	element x 0 or more times
$x+$	element x 1 or more times
$x?$	element x 0 or 1 time

Given the variety of CAMILA data models it is easy to see that for each case we can have more than one correspondence. For example, the following mapping could represent a translation scheme:

Translation Scheme	
SGML	CAMILA
x, y	tuple
x & y	tuple
$x \mid y$	alternative
$x*$	X-seq
$x+$	X-seq
$x?$	[X]

The above scheme is poor in some respects. For example $x+$ is being mapped into *X-seq* but this list should have one or more element. This can be defined through an invariant. The translation to CAMILA besides converting the types should add the necessary invariants to each case.

The relation between SGML and our system will be further explored in a forthcoming paper.

4 SOME EXAMPLES

In order to illustrate some of the advantages of the proposed approach, we present two examples using CAMILA .

4.1 MAIL

In this example we specify what a *unix mail message* is. Next the specified structure is used in some real processing.

To define the document type that describes a *unix mail message* we could write the following CAMILA specification:

```
MODEL mail

use "txt.cam"

TYPE
  header= SYM -> ANY;
  mail  = h:header
          b:TXT;
  env   = user:SYM
          date:ANY;
ENDTYPE

STATE e:env;
 e <- env('joao,"today");   /*inicial state*/
```

Mail is composed by `header` and `body`. The `body` is simply text. The `header` is a finite function from symbol to anything, where symbol is a token; in this case pertinent tokens are: `to`, `from`, `cc`, ...

Now we can write some functions over that type reflecting our knowledge about the behaviour of mail messages. For example, it may be stated that a mail message, in order to be considered corret, should have a `from` field and its `body` should not be empty. This can be writen in CAMILA as the following invariant:

```
inv_mail(a)=
   'from in dom(h(a)) /\
   (b(a) != "" \/ h(a)['subj] != "");
```

Following we specify a *mail reply*:

```
func reply(a:mail):mail
returns
  mail(['to -> h(a)['from],
        'subj -> strcat("re:",h(a)['subj]),
        'from -> user(e),
        'date -> date(e),
        'cc -> h(a)['cc]],
      < "In the last episode you said:" :
           <strcat("> ",x) | x <- b(a)>>);
```

To finish this example we reproduce a mail session in CAMILA (putting things to work), we begin creating a document of type `mail`:

```
ex<-mail(['to-> 'joao,
          'from -> 'peter,
          'subj ->"Test of the system",
          'cc -> 'jcr],
  < "dear Joao",
    "good luck with this" > ) ;
```

so that we can apply to that document (`ex` mail message) the function `reply`

```
re_ex <- reply(ex);
```

the document `re_ex` now has the value:

```
    mail(['to-> 'peter,
          'from -> 'joao,
          'subj ->"Re: Test of the system",
          'cc -> 'jcr],
  < "In the last episode you said:"
    "> dear Joao",
    "> good luck with this" > )
```

Now it is necessary to edit the body of the mail in order to continue the message. The function `txtedit` will do that task by:

- writing the `STR-seq` body to a file (`txtsave`)
- calling an external editor (Ex. `vi`)
- reading back a text (`txtload`) (using an external `txt2cam` format translator)

```
func txtedit(txt:TXT):TXT
returns do( txtsave("_tmp",txt),
            sh("vi _tmp" )),
            txtload("_tmp"));

func txtload(name:STR):TXT
returns
 let(f=popen(strcat("txt2cam ",name),"r"),
     t=readf(f),
     x=pclose(f)) in t;
```

Now it possible to edit `re_ex` body in order to continue the message and to finish the reply:

```
 re_ex.b  <- txtedit(b(re_ex));
```

4.2 Literate programming

In this section a naïve literate programming(Knuth, 1992) system is described.[2]

The main idea is to have a document type `lpt` (literate programming type) that is a list of elements which can be:

- titles (of document(tit) or section(sec))
- association of identifiers(id) with programs(pro)
- program(pro) – sequences of strings(STR) or program references(id)
- straight text strings(STR)

That document contains a program (to be extracted with `getprog` function) and a textual document (to be extracted with `getlatex`) typically a manual describing the program implemention and including the program.

```
MODEL lp
TYPE
 lpt = ele-seq;              list of elements
 ele = STR | pro | defi | id | sec | tit;
 pro=(STR | id)-seq;          program with id
 defi = i : id                  id definition
        v : pro;
 id  = SYM;                        identifier
 sec = STR;                     section title
 tit = STR;                    document title
ENDTYPE
```

Let `ex` be an example document (built using the implicit constructors of the language)[3]:

```
ex <- <
  tit("Example of literate prog"),
  sec("Stack - FAQ"),
   defi('main, <"main(){...}",
               "int S[20]; sp=0",
               'pop ,
               'push >),

  sec("pushing elements"),
   "to push elements",
   "you can use this function:",
   defi('push, <"void push(int x)",
               "{S[sp++]=x;}">),

  sec("poping elements"),
   "not yet available",
   defi('pop,  <"int pop(x)",
               "{/*to be continued*/}">)>;
```

Next we define the function `getprog` whose purpose is to extract a program(prog) from a literate programming text(lpt).

In the first step an index is built (function `mkindex`). The function `explode` is defined to make the recursive substitution of identifiers(id).

[2] The examples complete code (including the auxiliar functions not presented here because of space constrains) and other case studies can be obtained from the authors.

[3] A more WEB-like notation could be used based on a `webget` translator (easily built in PERL).

```
TYPE
  prog = STR-seq;          (prog with no id)
  index = id -> ele-pro;
ENDTYPE

func mkindex(t:lpt): index
return [i(x) -> v(x) | x<-t : is-defi(x)];

func getprog(t:lpt): prog
return  explode('main,mkindex(t));

func explode(i:id, d: index) : prog
pre i in dom(d)
returns CONC(
  < if(is-id(x)-> explode(x,d),
       else    -> <x>         ) |x <- d[i]>);
```

Let pex be the program extracted from ex:

```
  pex <- getprog(ex);
```

would assign to pex

```
  main(){...}
  int S[20]; sp=0
  int pop(x)
   {/* to be continued * /}
  void push(int x)
   {S[sp++]=x;}
```

To extract the document part(latex) of the literate programming text, we have to define the document type latex[4]:

```
/* micro Latex */
   latex =
      d : documentclasse   /* article*/
      t : tit               /* title */
      s : section-seq ;     /* body  */
   section =
      t : sec
      v : (STR | verbatim)-seq ;
   documentclasse = SYM ;
   verbatim = STR-seq;

func getlatex(t:lpt):latex
returns
 if (t is-<ti:se>->latex('article,
                        ti,
                        getsecList(ta)));
 ....
```

To create the latex part of ex:

```
 latex_ex <- getlatex(ex);
```

[4]In order to be useful, this example should also include a generate function that produce the actual LaTeXsyntax from the camila latex document type.

would assign to latex_ex

```
latex(
   article ,
   tit( Example of literate prog ),
   < section(
        sec( Stack - FAQ ),
        < verbatim( < main
                      main(){...}
                      int S[20]; sp=0
                      pop
                      push >)>)
     section(
        sec( pushing elements ),
        < to push elements
          you can use this function:
          verbatim( < push
                      void push(int x)
                      {S[sp++]=x;} >)>)
     section(
        sec( poping elements ),
        < not yet available
          verbatim( < ...
```

5 Conclusion

Along this paper we have discussed an approach to document processing we intend to develop further: *define document types and specify document manipulations under an algebraic system.* Types are described using the usual abstract data models plus a predicate that establishes type invariants. Documents are created, and processed as instances of a given type by means of function application. Those functions with type models define an algebra and documents can then be thought as algebraic terms.

Our proposal is based on the use of the algebraic system CAMILA, a general purpose constructive specification language and an environment for building and running program prototypes.

With this approach we gain in simplicity and conciseness. Moreover, we think that three other obvious advantages emerge from this method: the reusability of types and functions; the proofability of correctness, based on type invariant checking and validation of function calls (with respect to its signature); the refinement guidelines.

SGML was compared to other solutions and has been chosen as the external document description language to interchange documents with our system.

Two examples —definition and manipulation of Unix mail messages and literate programs— were presented for illustration of our approach, its style and its power.

Another topic that is currently under research is the use of *attributed abstract syntax trees* to store and manipulate documents under an algebraic approach.

The long term aim is to develop an automatic, or semi-automatic, translation process based on the systematic analysis of document types.

Bibliography

Barbosa, Luis and J.Joao Almeida. 1995. *System Prototyping in CAMILA*. University of Minho. Lecture notes for the system Design Course, Computer System Engineering, University of Bristol.

Cowan, D., E. Mackie, G. Pianosi, and G. d. V. Smit. 1991. Rita - an editor and user interface for manipulating structured documents. *Electronic Publishing, Origination, Dissemination and Design*, 4:125–150.

Feijs, L. and H. Jonkers. 1992. *Formal Specification and Design*. 35. Cambridge Tracts in Theoretical Computer Science.

Germán, Daniel M. and D. D. Cowan. 1995. Experiments with the z interchange format and sgml.

Goguen, J., J. W. Thatcher, and E. G. Wagner. 1978. Initial Algebra Approach to the Specification, Correctness and Implementation of Algebraic Data Types. In *Current Trends in Programming Technology*, volume IV. Prentice-Hall International.

Guttag, J. and J Horning. 1993. LARCH*: Languages and Tools for Formal Specification*. Springer-Verlag.

Harbo, Klaus. 1994. *CoST version 0.2 - Copenhagen SGML Tool*. University of Copenhagen.

Harper, R. and K. Mitchell. 1986. Introduction to Standard ML. Technical Report, University of Edimburgh.

Haxthausen, A. 1990. A Tutorial on RAISE. Technical Report RAISE/CRI/DOC/1-2-3-9, CRI A/S (Denmark).

Hendersen, P. 1984. me too: A Language for Software Specification and Model Building — Preliminary Report. Tecnhical Report, University of Stirling.

Knuth, Donald E. 1992. *Literate Programming*. Distributed by Unversity of Chicago Press. CSLI–27.

Lamport, Leslie. 1986. *LaTeX User's Guide and Reference Manual*. Addison-Wesley Publishing Company.

Megginson, David. 1995a. sgmlspl: a simple post-processor for sgmls and nsgmls. Technical report, Dep. English - Univ. Ottawa.

Megginson, David. 1995b. Sgmls.pm: a perl5 class library for handling output from the sgmls and nsgmls parsers. Technical report, Dep. English - Univ. Ottawa.

Nipkow, T. 1986. Non-Deterministic Data Types: Models and Implementation. *Acta Informatica*, (22):629–661.

Oliveira, J. N. 1990. A Reification Calculus for Model-Oriented Software Specification. *Formal Aspects of Computing*, (2):1–23.

Oliveira, J. N. 1992. Software Reification Using the SETS Calculus (invited communication). In *Theory and Practice of Formal Software Development*. BCS FACS 5th Refinement Workshop, London.

Sperberg-McQueen, C.M. and Lou Burnard. 1994. *Guidelines for Electronic Text Encoding and Interchange (TEI P3)*. Chicago: ACH/ACL/ALLC.

Turner, D. A. 1986. MIRANDA: A Non-Strict Functional Language with Polymorphic Types. *Jour. Comp. Sys. Sci.*, (19):27–44.

A CAMILA : a brief introduction

Parts of this appendix come from (Barbosa and Almeida, 1995) lecture notes where a more detailed overview of CAMILA can be obtained.

A.1 CAMILA philosophy and evolution

From school physics we got used to a basic problem solving strategy: *create a mathematical model, reason on it, calculate a solution*. The CAMILA approach is an attempt to make such a strategy available at the software engineering level. Based on a notion of *formal software component* it encompasses a set-theoretic notation, a prototyping environment, fully connectable to external applications and equipped with communication facilities, and an inequational refinement *calculus*.

CAMILA aims to be both a learning tool for Computer Science students and a working tool for software engineers. At the first level it provides a smooth way to programming. At the second a rigorous way to develop complex systems and to promote the use of formal methods in software industry.

CAMILA[5] was originally devised as a collection of interrelated support tools for teaching different parts of the Computer Science and Software Engineering curricula. The project affiliates itself, but is not restricted to, to the research in exploring Functional Programming as a *rapid prototyping* environment for formal software models, whose origin can be traced back to P. Hendersen's me too (Hendersen, 1984).

In the way, some new theoretical and technological results — namely a component classification and reification calculus and a notion of connectable high-level prototyping environment — were achieved and incorporated in the project.

As a working tool for software engineers it offers a simple set-theoretic notation and a fully connectable environment. As a learning tool supporting a Computer Science curriculum, it aims to be easy to understand and use, and to stimulate a kind of abstract and compositional reasoning which paves the way to sound methodological principles.

The CAMILA platform is organized around 5 main components:

- An executable (functional) specification language directly based on *naive* set theory.
- An inequational calculus (Oliveira, 1990; Oliveira, 1992) — SETS — for refining and classifying software formal models. In particular it enables the synthesis of target code programs by transformation of the initial specifications.
- A flexible rapid prototyping kernel which bears "full citizenship" at C/C++ programming level (C may call CAMILA services and CAMILA may also invoke external C functions). It is available at both UNIX, LINUX and MS/DOS operating systems and may provide services under X-WINDOWS or as a WINDOWS 3.1 DLL. Furthermore the prototyping environment provides a set of communication facilities to animate systems built by composition of independent and concurrent software components.
- A formal software components *repository* which catalogues available models and a compositional notation based on "software-circuit" diagrams (a shorthand for some piece of mathematics), suggestively resembling the conventional hardware notation.
- An approach to the specification and generation of structural Human-Machine Interfaces, independent of but mirroring the application semantics.

The CAMILA approach to programming technology claims to provide a smooth way to teaching and using (constructive) formal methods in software engineering. Its roots on functional prototyping of information models (Hendersen, 1984) has already been referred. Similar motivations may be found either in the research on *formal specification methods*, such as VDM, Z, RAISE (Haxthausen, 1990), COLD-K (Feijs and Jonkers, 1992) or LARCH (Guttag and Horning, 1993), or on *functional programming languages* such as ML (Harper and Mitchell, 1986) or MIRANDA (Turner, 1986).

In contrast with the former group one could stress the lighter notation of CAMILA, borrowed from set theory, and the direct correspondence to the prototyping language. But what is, to our knowledge, new is the associated calculus for model reasoning and refinement. On the other hand, CAMILA lacks the sophisticated interface and documentation management features available, for instance, in RAISE.

CAMILA, or at least its prototyping language, may also be compared with other functional languages which achieved a high degree of clarity and expressive power. Although some features of more elaborated languages (*eg*, effective polimorphism) are absent in CAMILA, we would point out as original features CAMILA's flexibility in being fully connectable to other "galaxies" of the computation universe and easily suited to different application domains.

[5] CAMILA is named after a Portuguese 19^{th}-century novelist — Camilo Castelo-Branco (1825 - 1890) — whose immense and heterogeneous writings, deeply rooted in his own time experiences and controversies, mirrors a passionate and difficult life.

A.2 The Camila Language

A Camila specification is a set of software components. Each one is a *model* that includes *type*, *function* and *state* definitions.

```
Model --> MODEL id
           TypeDef
           FunDef
           StateDef
          ENDMODEL
```

Where a type definition has the following form:

```
TypeDef --> TYPE
             ( id = TypeModel )*
             ENDTYPE
```

The basic data type models predefined in Camila are:

Data Models	Camila
Sets	X-set
Lists	X-seq
Finite functions	X $\longmapsto$ Y
Binary Relations	X $\longleftrightarrow$ Y
Alternatives	X | Y | ...
	[X]
tuples	T = X : A Y : B
Integers	INT
Strings	STR
Tokens	SYM
Universe	ANY

Camila also provides some other primitive types which do not bear a direct mathematical correspondence but are inherent to its programming environment.

A function definition has the following form:

```
FunDef --> FHeader FPredCond FState FBody
FHeader--> FUNC fid (ParamLst) : typeid
FPreCond--> PRE CondExp
FState --> STATE exp
FBody  --> RETURNS Exp
```

Finally, a state definition is written according to the rule:

```
StateDef  --> STATE sid : typeid
```

The basic collections of functions associated with Camila type constructors (*eg*, intersection or union of two sets, domain or range of binary relations, application or overwrite of mappings, concatenation of sequences and reduce operators, structure definition by enumeration or comprehension, etc.) are available as primitive functions in the language. So are the propositional connectives and quantifiers. To exemplify, a synopsis of some collections is presented above in the form of tables showing the Camila syntax, a brief informal description and the corresponding set theoretic notation.

(Finite) Functions — $X \longmapsto Y$

Camila	Description	Semantics
`dom(f)`	Domain	$\mathsf{dom}\, f$
`ran(f)`	Co-domain	$\mathsf{rng}\, f$
`f[x]`	Application	$f[x]$
`f/s`	Dom. restriction	$f\|s$
`f\s`	Dom. subtraction	$f \setminus s$
`f + g`	Overwrite	$f \dagger g$
$[\_ \mapsto \_, \ldots]$	Map. enum.	[...]
$[x \to e \mid x \leftarrow s : p]$	Map. compreh.	$[e\| \; x \in s \wedge p]$

Sequences — `X-seq`

Camila	Description	Semantics
`hd(s)`	Head	$\mathsf{hd}\, s$
`tl(s)`	Tail	$\mathsf{tl}\, s$
`nth(i,s)`	Elem. by pos.	$s(i)$
`s^r`	Concatenation	$s \frown r$
`<x:s>`	Appending	$<x> \frown s$
`CONC(s)`	concatenation	$s_1 \frown s_2 .. \frown s_n$
`elems(s)`	Set of elements	$\{x\| \; \underline{x \leftarrow s}\}$
`inds(s)`	Domain	$\mathsf{dom}\, s$
`plusq(s,f)`	overwrite	$s \dagger f$
`<e|x<-s:p>`	Seq. compreh.	$< e\| \; x \in s \wedge p >$
`o-orio(e,s)`	Distribut. form	