Theoretical Computer Science

ELSEVIER Theoretical Computer Science 199 (1998) 231–247

Algebraic specification of documents

José Carlos Ramalho*, José João Almeida, Pedro Henriques

Departamento de Informática, Universidade do Minho, Braga, Portugal

Abstract

According to recent research, nearly 95% of a corporate's information is stored in documents. Further studies indicate that companies spend between 6% and 10% of their gross revenues in printing and distributing documents in several ways: web and cdrom publishing, database storage and retrieval, and printing. In this context documents exist in different formats, from plain text files to internal database or text processor formats. It is clear that document reusability and low-cost maintenance are two important issues in the near future.

The majority of available document processors is purpose-oriented, reducing the necessary flexibility and reusability of documents. The problem of adapting the same text to different purposes gives rise to waste of time. For example you may want to have the same document as an article, as a set of slides, or as a poster; or you can have a dictionary document producing a book and a list of words for a spell checker. This conversion could be done automatically from the first version of the document if it complies with some standard requirements. The key idea will be to keep a complete separation between syntax and semantics. In this way, we produce an abstract description separating conceptual issues of document structure from those concerned with document use.

This note proposes a few guidelines to build a system to solve the above problem. Such a system should be an *algebraic based environment* in order to provide facilities for

- definition of document types;
- specification of functions over document types; and
- definition and handling of documents as algebraic terms.

Our approach (*rooted in the tradition of constructive algebraic specification*), allows for a homogeneous environment to deal with operations such as *merging* documents, *converting* formats, *translating* documents, *extracting different kinds of information* (to setup 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. © 1998 — Elsevier Science B.V. All rights reserved

Keywords: Document processing; Algebraic specification; SGML; CAMILA

* Corresponding author. E-maile: jcr@di.uminho.pt.

PII S0304-3975(97)00274-0

1. Introduction

A *document* is a collection of pieces of text – pure character strings – organized according to a specific structure. Its information content can be viewed as a message to be delivered (to someone), and its structure is defined in order to emphasize some special parts of that message, and in general, to improve its transmission process.

When dealing with documents on digital support, it is very important to make their structure explicit (to allow for automatic structure recognition and validation). That is what text processing and word processing systems do, each system in its own way.

In order to make a document's structure explicit, additional information must be interspersed with the natural text of the document. This added information, called *markup*, serves two purposes:

- separating the logical elements of the document; and
- specifying the processing functions to be performed on those elements.

The tags added to the text (markup), form the lexicon of a language, a *markup language* [8, 19].

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 nothing more than a typical language processor where at least two languages are involved – the markup language, used to define the document structure, and the language(s) used to express the information content of the document.

(*Model based*) *algebraic specification* 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 a many sorted algebra [5] (or relational structure [14]) for a given signature (i.e. a set S of sorts and an S^*-indexed set of operation symbols). The model consistently assigns a set to each sort symbol and a function (or relation) to each operation (or predicate) 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 user-defined. 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 method can be useful to specify documents and tools and to 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. Since we already know how to refine algebraic functions into procedural programs [15, 16], that mapping will enable one to formally obtain implementations from specifications and prototypes.

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

2. The proposed algebraic approach

Document definition is an old known problem.

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

It should be possible to formally describe the behavior of the tools used to manipulate documents. Furthermore, those tools should help us to guarantee

- document structural correctness – have the right components according to text purpose,
- invariant preservation – where invariants are some defined constraints which are 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 of the same document.

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

Example 1 (*Document reuse*). A dictionary can be printed. However, its definition should not be tied to pagination, because that would clutter, if not even disable the possibility of reusing it for other operations, such as its conversion into an electronic hyper-text.

In the sequel, we elaborate on these ideas resorting to the CAMILAframework. The close resemblance of the specification language and elementary set theory makes the notation almost self-explanatory. However, the reader is referred to [2] for a complete account and to the appendix a for a brief overview.

2.1. Document-type definitions

The rules that define the possible structures for a given kind of document form the *document-type definition* of that type of document. Therefore, it is a step to the notion of document correctness.

In CAMILA a type specification involves the definition of:
- its carrier set
- an invariant – a boolean-valued function that restricts the carrier set structure to cope with specific semantic requirements.

Example 2 (*Electronic mail*). When dealing with electronic mail system specification, we must define the carrier set of the `mail` sort. The carrier set definition is

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

(1) mapping between identifiers and strings.
(2) a list of strings (lines of text).

The following invariants guarantee that messages have nonempty 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 satisfy the invariant.

The structure of a document is also a good guide to building 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 can serve the following goals:
- describe document manipulation (such as translation between different formats);
- describe the behavior (or intended behavior) of existing tools;
- support future tools and (document) types;
- build documentation of tools and formats.

In the adopted framework, the definition of a function comprises the following steps:
- Definition of its domain and co-domain: the enumeration of the sorts for its arguments and the expected sort for the result (we call it *the function signature*).
- Definition of a precondition (predicate over arguments and state) that has to be evaluated to *true*, so that the function can be applied
- Definition of a returned value, whenever the function is applicable (the precondition holds).
- Definition of how to update the state, upon function application.

Function specifications are an important step in the definition of system (document and tools) correctness. From this point of view, a *function definition* must guarantee:
- That the invariant of the returned value-type evaluates to *true*
- That the invariant of the computed state evaluates to *true*.
- That the precondition of every function used is true

and a *function application* must
- be well typed,
- verify the corresponding precondition.

The basic collections of operators associated to CAMILA-type constructors (e.g., union, intersection of two sets, domain or range of binary relations, application or overwrite of mappings, etc.) are available as primitive functions in the language. So are the propositional connectives and the first-order quantifiers.

The availability of all the repertoire of CAMILAoperators and the guidelines offered by the type model, as exemplified above, greatly simplify the task of defining new document processing functions.

3. The algebraic system

It is quite clear that an algebraic system is of limited expressivity, concerning the reality of document electronic interchange. This entails the need for another layer intended to establish a bridge between the algebraic system and the outside world of documents. A format (or set of formats) is to 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 be easy to parse and generate. The layer will incorporate a parser/translator for the chosen formats.

Fig. 1 exemplifies the idea of the intended system in more detail, where
- f1 Denotes a CAMILAfunction that receives two documents as arguments and produces a new one.
- f2 Denotes a CAMILAfunction that transforms a document into another.

External processing using external tools, e.g. accepting a format FMT1 and building a document in format FMT2, is modeled by defining an `exportFMT1` and a `importFMT2` functions (see `txtedit` in mail example, Section 4.1). Looking at the current scene, there are some strong candidates to be considered as an input/output format to/from our system such as LaTeX [10], Word or SGML [18]. On the other hand, a closer look at those formats shows that Word is not a good choice because it does not have a visible structure and its format (under a private 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 to process documents written in their formats.

Though one major difference comes up, LaTeX is too tied up to format the typographic aspects, whilst SGML is not. Besides this, 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

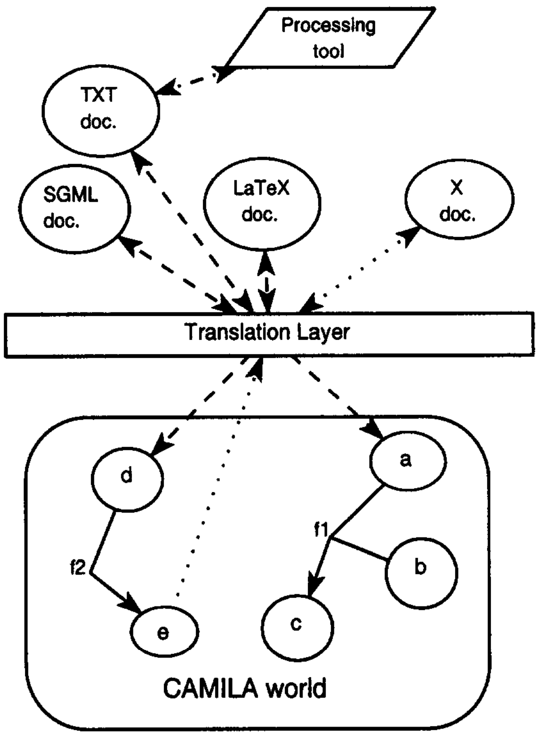

Fig. 1. System architecture

NSGMLS, SP, JADE, RITA [3], CoST [6] and sgmlpl [13, 12] 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 specify 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 documents by 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 type (or class) of documents.

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

When creating a DTD, structural elements and the way they are related to each other 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, in the following mail message:

Example 3 (*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 corresponding DTD may be specified as:

Example 4 (*Mail DTD*).

```
  <!ELEMENT mail   - - (header,body)>
/* mail is composed of header and body */
  <!ELEMENT header - - (from,to)>
/* header is composed of 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 a faithful interpretation of SGML into CAMILA(data models) exists. In the next section we will show that it is possible to model SGML constructs with CAMILAData Models. This will enable us to define a translation function from SGML to CAMILA.

3.2. SGML ↔ CAMILA*data models*

SGML is a very simple, structure-oriented language, so it should not be difficult to create a correspondence between its features and appropriate CAMILAdata models.

An SGML specification is composed of a series of ELEMENT declarations. Each ELEMENT corresponds to a structural element of the document and is defined as text

Table 1

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

Table 2

Translation scheme

SGML	CAMILA
x, y	Product
x & y	Product
$x \mid y$	Disjoint union
$x*$	X-seq
$x+$	X-seq
$x?$	[X]

or as being a combination of other elements. SGML has a few operators to specify relations between elements given in Table 1.

Given the variety of CAMILAdata models, it so happens that more than one of them could be chosen to correspond to each of the features listed above. For example, the mapping shown in Table 2 could represent a translation scheme:

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 by means of 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 is further explored in [17].

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, we use the specified structure to specify 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  /* operating system */
           date:ANY; /* environment      */
ENDTYPE

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

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

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

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

In the following we specify a *mail reply* function:

```
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. We begin by creating a document of type `mail`:

```
ex<-mail([to-> joao,
          from -> peter,
          subj ->"Testing",
          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 allow the user to edit the body of the mail in order to continue the message. The function txtedit will do that task by

- writing the message 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)) 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));
```

In the example, sh("vi ...") executes an external command (vi editor); popen ("txt2cam file","r") opens a channel (pipe) to read the output of an external command (txt2cam format translator); readf(channel) reads an expression from a file.

The last example resorts to the use of CAMILA interface functions with "the outside word". In fact, the CAMILA prototyping environment provides mechanisms to invoke external C functions and to read and write to external commands [1]. This feature seems to be important.

4.2. Literate programming

In this section a naïve literate programming [9] system is described. [1]

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)),

[1] The complete examples (including the auxiliary functions not presented here because of space constraints) and other case studies can be obtained from the authors.

- program definitions, associations of identifiers(id) with programs(pro),
- programs(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 implementation 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): [2]

```
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("popping 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).

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

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

```
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`:[3]

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

[3] In order to be useful, this example should also include a generate function that produce the actual LaTeX syntax from the CAMILA `latex` document type.

```
    documentclass = 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);
```

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( popping elements ),
        < not yet available
          verbatim( < ...
```

5. Conclusions

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 of as algebraic terms.

Our proposal is based on the use of the CAMILA platform, 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, three other advantages emerge from this method: the reusability of types and functions; the correctness proof, 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.

Acknowledgements

We would like to thank the precious comments and suggestions from our anonymous referees that were very helpful to improve this paper. We are also grateful to Luis Barbosa for the profitable discussions and for the CAMILA material incorporated in the appendix. Thanks are due to J.N.I.C.T. for the grant under which this work is being developed.

Appendix A. CAMILA: A brief introduction

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[4] was originally devised as a collection of interrelated support tools for teaching different parts of the computer science and software engineering curricula. The

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

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 Hendersen's me too [7].

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.

The CAMILA platform is organized around five main components:

- An executable (functional) specification language directly based on *naive* set theory.
- An inequational calculus [15, 16] – 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 [11].

A.2. Notation overview

CAMILA's basic construction is a notion of *formal software model* including *type*, *function* and *state* definitions according to the following syntax:

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

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

Mathematically, this stands for a *relational structure*, i.e. a local hidden *state* space equipped with a set of *operations* which can access and modify its current value. State spaces, as well as other data sorts involved, are described by (possibly recursive) set-theoretical expressions and may be subjected to *invariants*, i.e. structural properties intended to be preserved by every operation. Operations are defined as functions or relations over the state space and their specific parameters. Hence, a CAMILA specification resembles what is called a *model* in the VDM meta-language or a *scheme* in Z.

Sorts and operations are described in the centenary notation of *naive* set theory, as arising from the basic constructs of the category of sets. Therefore, *cartesian product* (expressed by juxtaposition and optional labelling of each factor; example: `id1:A id2:B`), *coproduct* (expressed by `A|B`), also called disjoint union, and *exponentiation*, or function space, are the basic set constructors from which derived ones can be defined. Those include, for finite A and B,

`A-set`	*subsets*
`A <-> B`	*binary relations*
`A -> C`	*mappings*
`A-seq`	*sequences*

as well as the "null" alternative (`[A]`), and recursive definitions in the form $X = \mathbb{F}\ X$, where $\mathbb{F}$ is a set expression involving the above constructs.

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 syntax:

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

Finally, a state definition is written according to the syntax

```
StateDef --> STATE id : type
```

The state identifier *id* will be used whenever one has to access or modify the state.

The basic collections of functions associated with the CAMILA-type constructors are provided in the language. To exemplify, subsets of the mappings and sequences algebras

Table 3
Mappings — `X -> 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	$f \mid s$
`f\s`	Dom. subtraction	$f \setminus s$
`f+g`	Overwrite f by g	$f \dagger g$
`[ _->_, ... ]`	Map. enum.	[...]
`[x->e|x<-s:p]`	Map. compreh.	$[x \mapsto e \mid x \in s \wedge p]$

Table 4
Sequences — `X-seq`

CAMILA	Description	Semantics
`hd(s)`	Head	$\mathtt{hd}\,s$
`tl(s)`	Tail	$\mathtt{tl}\,s$
`s[i]`	Elem. by pos.	$s(i)$
`s^r`	Concatenation	$s \frown r$
`<x:s>`	Appending	$\langle x\rangle \frown s$
`CONC(s)`	concatenation	$s_1 \frown s_2 .. \frown s_n$
`inds(s)`	Domain	$\mathtt{dom}\,s$
`<e|x<-s:p>`	Seq. compreh.	$\langle e \mid x \in s \wedge p\rangle$

are presented in Tables 3 and 4 showing the CAMILA syntax, a brief informal description and the corresponding set theoretic notation.

References

[1] J.J. Almeida, L.S. Barbosa, From prototypes to implementations and Back, Tech. Report DI/INESC-94-10-2, INESC (U. Minho), 1994.

[2] L. Barbosa, J. Joao Almeida, System Prototyping in CAMILA, University of Minho, 1995, Lecture Notes for the System Design Course, Computer System Engineering, Uni. of Bristol.

[3] D. Cowan, E. Mackie, G. Pianosi, G.d.V. Smit, Rita – an editor and user interface for manipulating structured documents, Electronic Publishing Origination Dissemination Design 4 (1991) 125–150.

[4] D.M. Germán, D.D. Cowan, Experiments with the z interchange format and sgml, 1995.

[5] J. Goguen, J.W. Thatcher, E.G. Wagner, Initial algebra approach to the specification, correctness and implementation of algebraic data types, Current Trends in Programming Technology, vol. IV, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[6] K. Harbo, CoST version 0.2 – Copenhagen SGML Tool, University of Copenhagen, 1994.

[7] P. Hendersen, me too: a language for software specification and model building – preliminary report, Tech. Report, Univ. of Stirling, 1984.

[8] E. Herwijnen, Practical SGML, Kluwer Academic Publishers, Dordrechit, 1994.

[9] D.E. Knuth, Literate Programming, Distributed by Univ. Chicago Press, 1992, CSLI-27.

[10] L. Lamport, LaTeX User's Guide and Reference Manual, Addison-Wesley, Reading, MA, 1986.

[11] M. Martins, Métodos Formais na Concepção e Desenvolvimento de Sistemas Interactivos, Ph. D. Thesis, Univ. do Minho (DI), 1995.

[12] D. Megginson, sgmlspl: a simple post-processor for sgmls and nsgmls, Tech. Report, Department of English, Univ. Ottawa, 1995.

[13] D. Megginson, Sgmls.pm: a perl5 class library for handling output from the sgmls and nsgmls parsers, Tech. Report, Department of English, Univ. Ottawa, 1995.

[14] T. Nipkow, Non-deterministic data types: models and implementation, Acta Inform. (22) (1986) 629–661.

[15] J.N. Oliveira, A reification calculus for model-oriented software specification, Formal Aspects Comput. (2) (1990) 1–23.

[16] J.N. Oliveira, Software reification using the SETS calculus (invited communication), Theory and Practice of Formal Software Development, BCS FACS 5th Refinement Workshop, London, 1992.

[17] J.C. Ramalho, J.J. Almeida, P.R. Henriques, Document semantics: two approaches, in: SGML'96: Celebrating a decade of SGML, Sheraton-Boston Hotel, Boston, USA, November 1996.

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

[19] B. Travis, Dale Waldt, The SGML Implementation Guide, Springer, Berlin, 1995.