PLUSS

A LANGUAGE FOR STRUCTURED SPECIFICATIONS

ASSPEGIQUE

AN ENVIRONMENT FOR FORMAL SPECIFICATION

M-C. Gaudel

M. Bidoit, B. Biebow, C. Choppy,

S. Kaplan, A. Mauboussin, F. Voisin

YET ANOTHER SPECIFICATION LANGUAGE?

PLUSS is the result of several experiments in writing large specifications

- using algebraic data types

- in collaboration with people from industry

<u>STARTING POINTS</u>

- ASL primitives (Wirsing & Sanella)
- Some software tools *parser (CIGALE), symbolic evaluator (SPECTRAL)....*
- PERQ workstations

<u>CASE STUDIES</u>

- Recorders management in E10S *(a telephone switching system)*
- CSE *Electronic Sattelit Concentrator (together with Petri Nets)*
- PASCAL into P-Code Compiler
- ADA Data Types
- Level 2 & 3 protocols in CCITT no 7

<u>CURRENT RESEARCHES</u>

- PLUSS*: Specification Language*
- ASSPEGIQUE: *Integrated Specification Environment*
- MAIA: *Lisp-Prolog Workstation*

LARGE SOFTWARE PROJECTS AND FORMAL SPECIFICATIONS

OPEN PROBLEMS (some of!)

* STRUCTURING & MODULARIZING SPECIFICATIONS

* COEXISTENCE OF VARIOUS FORMALISMS *(f.i. Algebraic spec. + Petri Nets)*

* IMPACT ON OTHER STEPS OF THE DEVELOPMENT PROCESS *(f.i. Testing)*

- Hierarchical Specifications are easy to understand. They are difficult to design.

- When designing a specification you want to modify the class of models

- When using a specification you must not modify the class of models

REMARKS

- A Specification describes a <u>class</u> of implementations (i.e. Algebras)
- When you read a specification you assume that only finitely generated algebras are considered.
- When you write a specification you assume that <u>not</u> finitely generated algebras are considered.

A TOUR OF PLUSS in one slide

BASED ON ASL PRIMITIVES

specif **use**

proc } *parameterized specifications*

param

draft **enrich** **union**

renaming

visibility rules

observe

signatures

```
specif TABLE =
  use ELEMENT

    sort Table
    operations
      empty-table :    -> Table ;
      insert _into _: Element × Table -> Table ;

      _is-in _: Element × Table -> Bool ;
      remove _from _: Element × Table -> Table;

    gen empty-table, insert _into _

    axioms
              % definition of is-in %
      e is-in empty-table = false ;
      e is-in (insert e' into t) = (e is e') or (e is-in t) ;
              % definition of remove %
      remove e from empty-table = empty-table ;
      e is e' = false => remove e from (insert e' into t) =
                          insert e' into (remove e from t);
      e is e' = true => remove e from (insert e' into t) =
                          remove e from t
      where
      e, e' : Element ;
      t : Table
end TABLE
```

The TABLE specification

Example of *term* :

insert e into t

Example of *ground term* :

remove a from insert b into insert a into empty-table

```
specif COMPARABLE-TABLES =
  use TABLE

    operation
      _is-included-in _: Table × Table -> Bool
              % tests inclusion of tables %

    axioms
      empty-table is-included-in t = true ;
      (insert e into t) is-included-in t' =
              (e is-in t') and (t is-included-in t')
      where
      t, t' : Table ;
      e : Element
end COMPARABLE-TABLES
```

Example of use of the TABLE specification

```
proc PTABLE(ITEM) =

  sort Ptable
  operations
    empty :    -> Ptable ;
    add-entry : Item × Ptable -> Ptable ;

    suppress : Item × Ptable -> Ptable ;
    _belongs-to _: Item × Ptable -> Bool ;

  gen empty, add-entry

  axioms
    i belongs-to empty = false ;
    i belongs-to add-entry(i',t) = (i eq i')
                                   or (i belongs-to t) ;
    suppress(i, empty) = empty ;
    i eq i' = true => suppress(i, add-entry(i', t)) =
                          suppress(i, t) ;
    i eq i' = false => suppress(i, add-entry(i', t)) =
                          add-entry(i', suppress(i, t))
    where
    i, i' : Item ;
    t : Ptable
end PTABLE
```

Specification of Parameterized Tables

```
param ITEM =
  use BOOL

   sort Item
   operation
     _eq _: Item × Item -> Bool

   axioms
             % eq is an equivalence %
     i eq i = true ;
     i eq j = j eq i ;
     i eq j = true & j eq k = true => i eq k = true
     where
     i, j, k : Item
end ITEM
```

Specification of the Properties Required for the Arguments

```
specif TABLE =
  PTABLE(ELEMENT)

    renaming Ptable into Table,
          empty into empty-table,
          add-entry into insert _into _,
          suppress into remove _from _,
          _belongs-to _ into _is-in _
end TABLE
```

Making the TABLE Specification from the Parameterized One

```
specif SPEC export s, op, SPEC1 =
  use SPEC1, SPEC2

    sorts s, s'
    operations
      op : ... ;

    axioms ...

end SPEC
```

```
specif SPEC forget s', SPEC2 =
  use SPEC1, SPEC2

    sorts s, s'
    operations
      op : ... ;

    axioms ...

end SPEC
```

```
specif RESTRICTED-TABLE forget insert _into_,
                               remove_from_=
  use TABLE

end RESTRICTED-TABLE
```

Visibility Clauses

```
specif CONTINUUM =
  use MESSAGE, INTEGER
    sort Board, Erroneous-Message
    operations
      new-board : -> Board ;
      add-new-message : Message * Board -> Board ;
      erase-message : Int * Board -> Board ;
      read : Int * Board -> Message U Erroneous-
      Message;
      size : Board -> Int ;
      %the following operations correspond to error
      messages%
      erased-message : -> Erroneous-Message ;
      no-such-message : -> Erroneous-Message ;
    gen new-board, add-new-message, erase-message,
      erased-message, no-such-message
    axioms
      size(new-board) = 0 ;
      size(add-new-message(m, b)) = size(b) + 1 ;
      size(erase-message(n, b)) = size(b) ;

      read(n, new-board) = no-such-message ;
      n is size(b) = true =>
        read(n, add-new-message(m, b)) = m ;
      n is size(b) = false =>
        read(n, add-new-message(m, b)) = read(n, b) ;
      n is n' = true =>
        read(n, erase(n', b)) = erased-message ;
      n is n' = false =>
        read(n, erase(n', b)) = read(n, b) ;
    where
      b : Board ;
      m, m': Message ;
      n : Integer ;
end CONTINUUM
```

specif USER-CONTINUUM
export MESSAGE, INTEGER, Board, Erroneous-Message, add-new-message, read, size, erased-message, no-such-message
= CONTINUUM
end USER-CONTINUUM

specif CHAIRMAN-CONTINUUM =
use CONTINUUM, USER, PTABLE(USER)
%USER defines the attributes of users. Besides this specif uses tables of users which are renamed as users-lists.
renaming Ptable **into** Users-list; ...
end CHAIRMAN-CONTINUUM

VISIBILITY

idea (cf LARCH / Guttag & Horning)

use the rules of the "target" Programming Language

⇓

soon

ADA - PLUSS

CHILL - PLUSS

CLU - PLUSS

...

AN EXAMPLE
A SUBSET OF
THE UNIX FILE SYSTEM

```
draft FILE-T =
  use NAME, TEXT

    sort File
    operations
      name _: File -> Name ;
      content _: File -> Text ;
      <_, _> : Name × Text -> File

    axioms
      name <n, t> = n ;
      content <n, t> = t ;
      where
      n : Name ;
      t : Text
end FILE-T
```

File of Text : Draft

```
draft FILE-B =
  use NAME, BINCODE

    sort File
    operations
      name _: File -> Name ;
      content _: File -> Bincode ;
      <_, _> : Name × Bincode -> File

    axioms
      name <n, c> = n ;
      content <n, c> = c ;
      where
      n : Name ;
      c : Bincode
end FILE-B
```

File of Binary Code : Draft

```
draft FILE-E =
  use NAME

    sort File
    operations
      create : Name -> File ;
      name _: File -> Name

    axioms
      name create(n) = n
      where
      n : Name
end FILE-E
```

Empty File : Draft

```
draft FILE =
  enrich FILE-E, FILE-T, FILE-B by

    operation
      file _: File -> {empty, englishtext, executable}

    axioms
      file create(n) = empty ;
      file <n, t> = t ;
      file <n, c> = c
      where
      n : Name ; t : Text ; c : Bincode
end FILE


specif FILE from FILE =
    gen < _, _>, create

    % possibly some new operations can be added here %

end FILE
```

Making the FILE specification from Drafts

```
draft FOREST =
  use FILE, NAMELIST

    sort Directory, Forest
    operations
      name _: Directory -> Name ;
      content _: Directory -> Forest ;
      < _, _> : Name × Forest -> Directory ;
      file _: Directory -> {directory} ;
      φ :   -> Forest ;
      _•_: Directory × Forest -> Forest ;
      _•_: File × Forest -> Forest ;
      name-list-of : Forest -> Namelist ;
      .......

    axioms
      name <n, F> = n ;
      content <n, F> = F ;
      file <n, F> = directory;
      name-list-of (φ) = Λ ;
      name-list-of (d•F) = (name d)•name-list-of (F) ;
      .......
      where
      n : Name ; F : Forest ;
      d : Directory
end FOREST
```

Specification of Directories

```
draft SYSTEM =
  use PATH
  enrich FOREST by

    sort System
    operations
      root :    -> System ;
      mkfile : System × Path × File -> System ;
      mkdir : System × Path × Name -> System ;
      _: System -> Directory ;
    % This is a coercion: a system is a directory and
      all the operations on directories can be applied
      to systems. %
      ls : System × Path -> Namelist ;
      .......

    axioms
      p is p' = true => ls(mkfile(s,p,f),p') = (name f)•ls(s,p');
      p is p' = false => ls(mkfile(s,p,f),p') = ls(s,p');
      p is p' = true => ls(mkdir(s,p,n),p') = n•ls(s,p');
      p is p' = false => ls(mkdir(s,p,n),p') = ls(s,p');
      .......
      where
      p, p' : Path ; s : System ;
      f : File ; n : Name
  end SYSTEM
```

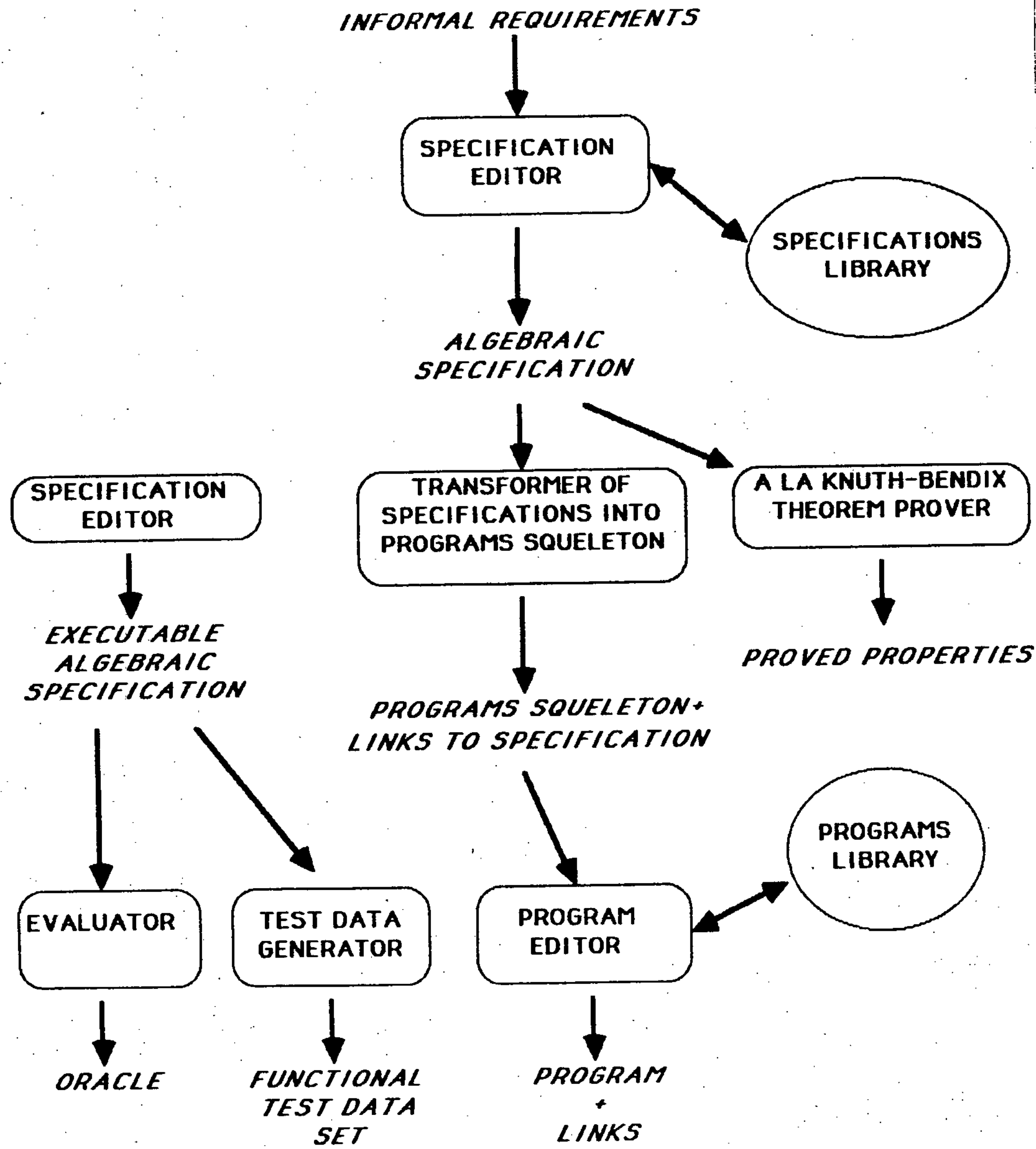
Specification of a File System: Draft

```
specif SYSTEM from SYSTEM =
    gen root, mkfile, mkdir for System ;
        ϕ, _•_for Forest;
        <_, _> for Directory;

    % operations such that rm or rmdir
      can be specified here. %

end SYSTEM
```

Making the FILE SYSTEM specification from the Draft



The design of ASSPEGIQUE

Two main motivations:

* Tools to make experiments on various theories, to test them.

* Tools to write and handle large specifications.

Ease of use, flexibility, user-friendly interfaces

* on-line documentation and help
* uniform multi-window interfaces
  * WINNIE
  * specific windows for the display of error messages, information ...
* graphic interfaces
  * pop-up menus, pointing devices
  * graphic representation of signatures, terms, proof steps, relations between specifications ...
* mix fixed syntax is allowed for operation symbols CIGALE

Identification of the tools required

. Hierarchical library management tool

. CIGALE : a tool for interactive and incremental grammar construction and expressions parsing.

Kernel specification environment :

Editor → Compiler → Symbolic Ev...

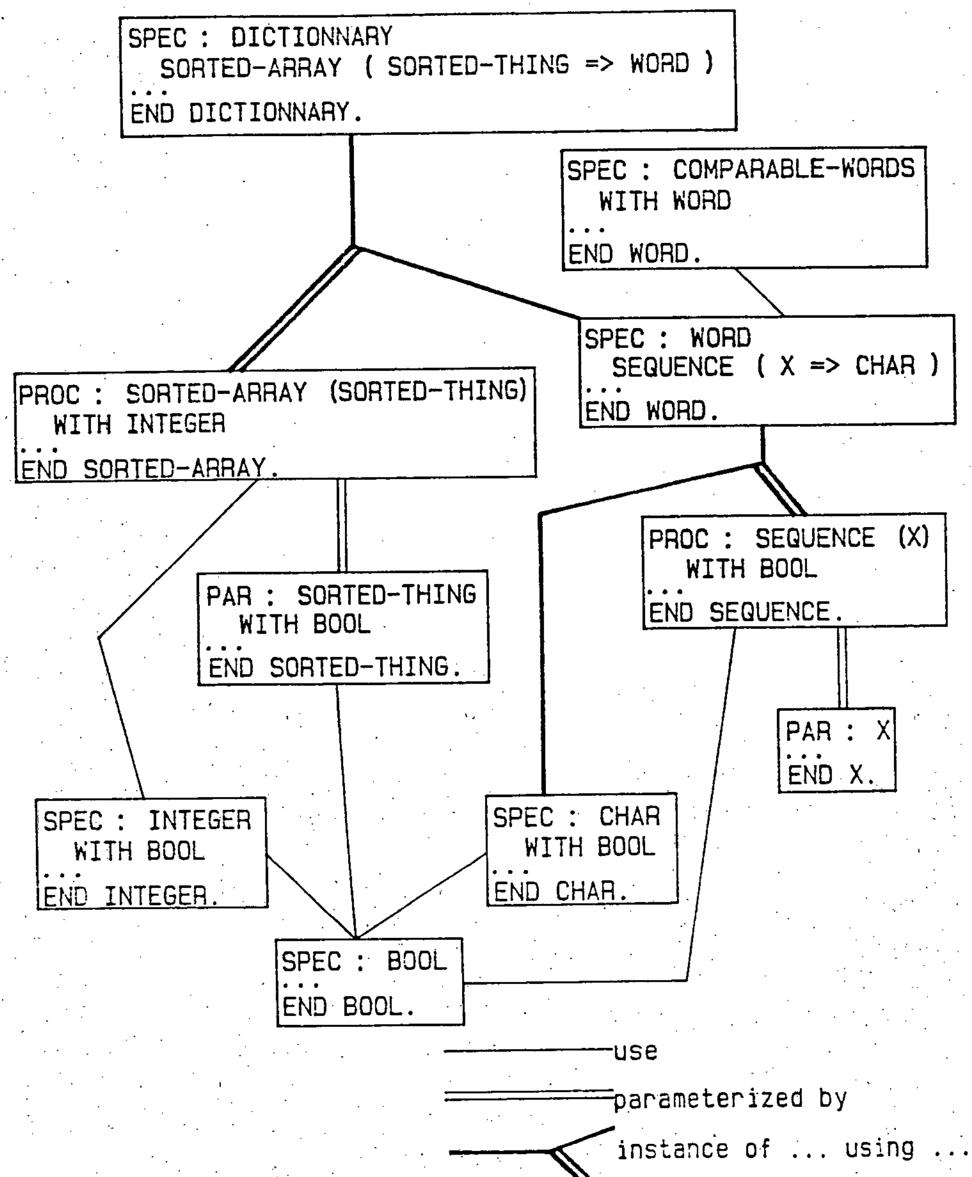
2nd Level specification environment :

. Modification and reuse tools

. Interactive debugger

. Theorem proving tools

...



RELATIONS BETWEEN SPECIFICATIONS

2 Note: There are two levels of syntax!
→ the specification language level (see the editor.
→ the term level in the axioms
↳ CIGALE

- CIGALE -

• Syntax of the operators must be free
→ enhance flexibility, legibility, correctne

"mixfix" syntax à la OBJ, with "_" to design the position of an operand.

e.g.: push_ onto_ : data stack → stack

• Incremental construction of the language
→ reuse of specifications, debugging of specifications

interactively add / delete operators

CIGALE (cont.)

- Modularity
  - → notion of a "current parsing environment"
  - → interactively add/delete (sub-) languages.

- Coercion, overloading and ambiguities

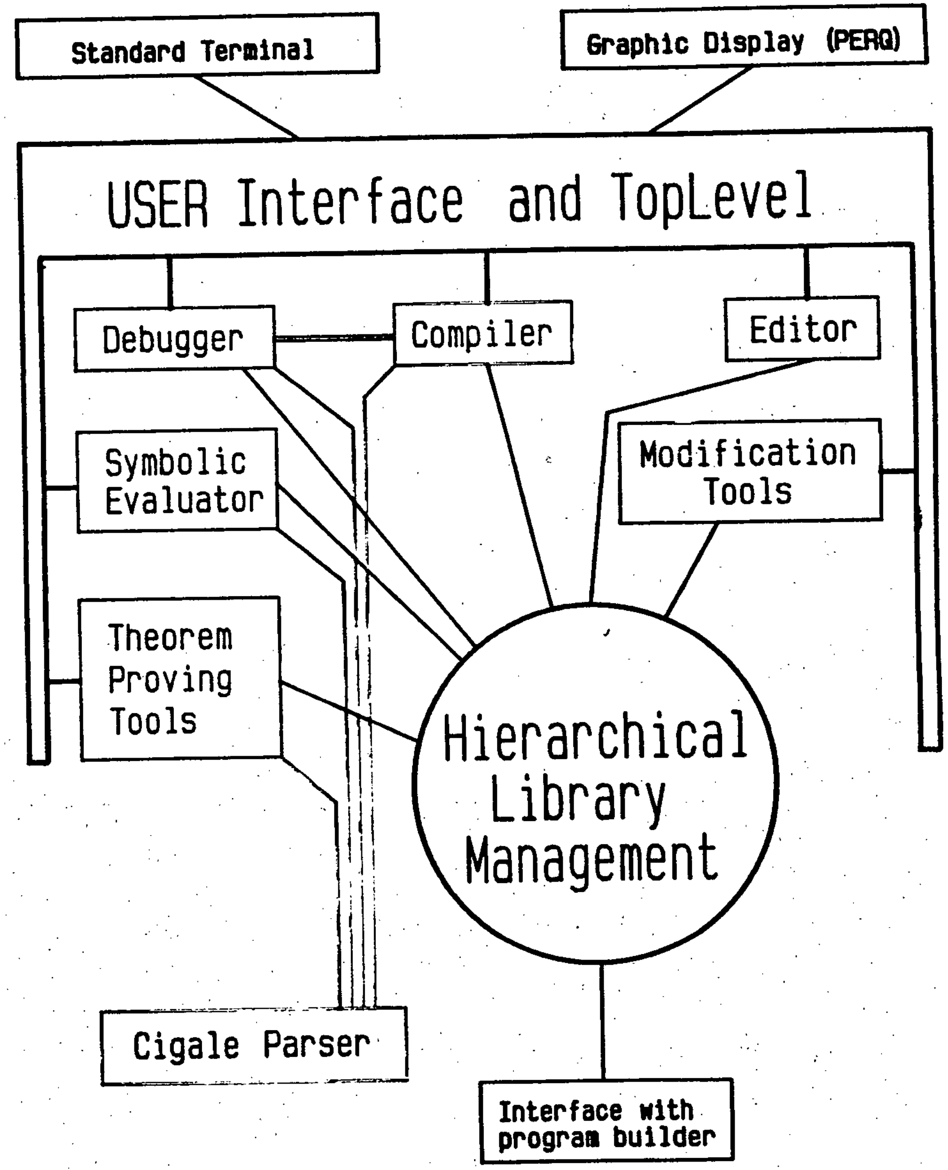
e.g.: _ : integer → monomial
_ : monomial → polynomial

or push _ onto _ : data stack → stack
append _ to _ : data file → file
x : → data

push x onto empty
append x to empty

useful ... and even necessary for variables!

ambiguities ⇒ "most natural analysis"



The specification editor

* syntax-directed

  with two concrete views:

  - a textual view
  - a graphic view

Internal representation is hidden from the user, editing is performed at the concrete level!

- at the textual level: like usual text editing, with syntax checks and special purpose functions.
- at the graphic level: pointing device (mouse) and pop-up menus.

* The editor produces an intermediate internal representation, updates library informations.

* Special facilities to interrupt and resume an editing session are provided.

The specification compiler

* computes the grammar associated to an elementary specification.

$$gram(spec) = \bigcup_{spec_i \text{ required by } spec} gram(spec_i) + \Sigma_{spec}$$

→ strongly [illegible] on CIRALE

* parses the axioms

* creates an internal representation and updates library information

? consistency of a specification cannot occur before required specifications are themselves compiled.

The internal representation produced by the compiler is the form that will be directly used by the other tools: symbolic evaluator, theorem proving tools, ...

The debugger

is loaded when errors are detected during compilation

allows to interactively debug axioms

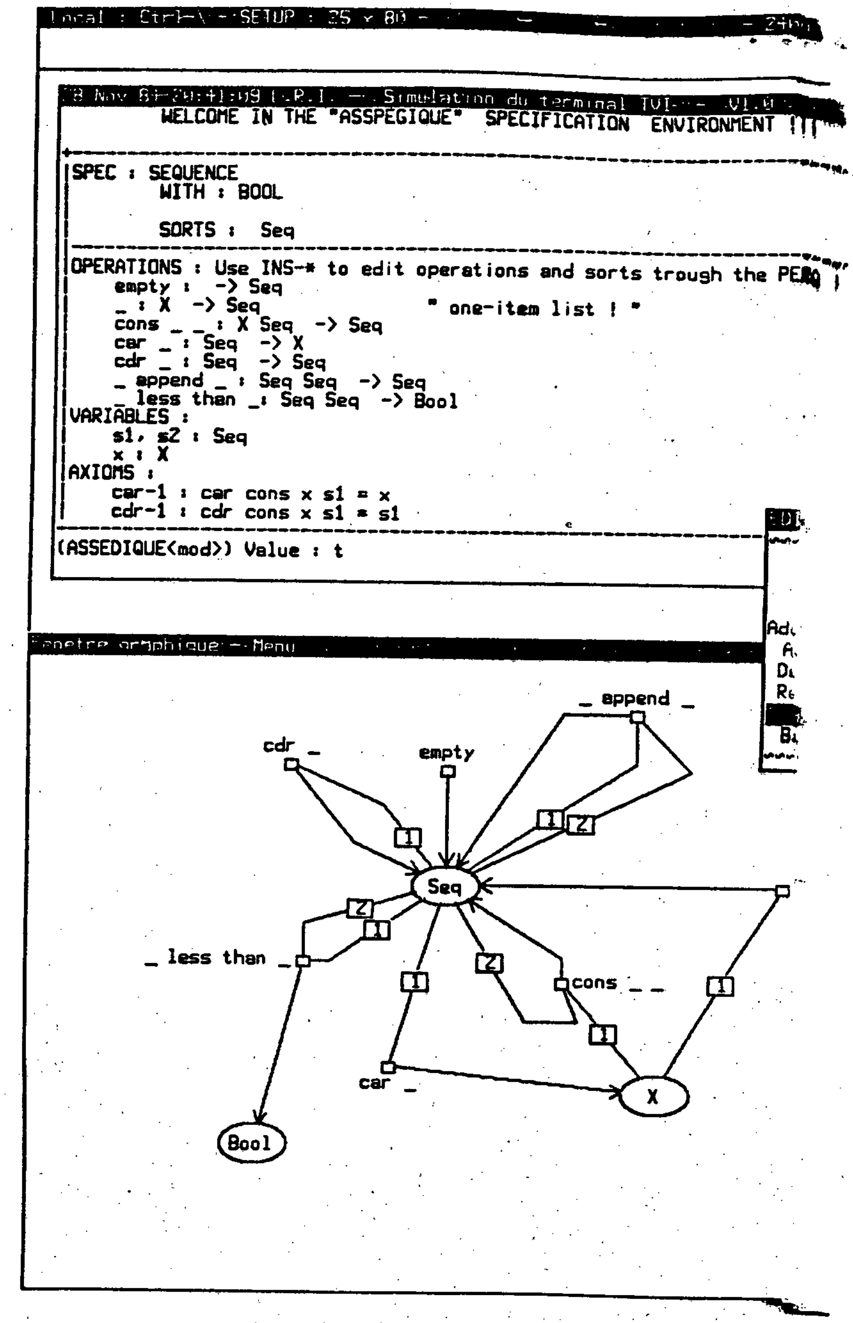
term rewriting system (computes the canonical form of an expression w.r.t. the axioms).

allows conditional axioms

takes parameterization into account.

Theorem proving tools:

- Generator induction in "fairly" specified equational theories
  (extension to conditional ones is investigated)
- will be used to verify correctness of
  - parameter passing
  - implementations



CONCLUSION

ASSPEGIQUE is implemented in Franz lisp under VAX/UNIX

([illegible]

[illegible]

[illegible]

→ still under development

→ evaluation and collaboration with industrial people will provide a firm basis to further developments