ASSPEGIQUE : An integrated environment for algebraic specifications

Michel BIDOIT & Christine CHOPPY

Laboratoire de Recherche en Informatique
Université de Paris-Sud
Bâtiment 490
91405 Orsay - Cedex, FRANCE

ABSTRACT

In this paper, we describe ASSPEGIQUE, an integrated environment for the development of large algebraic specifications. We first describe the underlying specification language, PLUSS-E, based on the specification-building primitives of ASL and E,R-algebras, a formal framework for exception handling. We then describe the design and organization of the specification environment. This environment allows the user to introduce specifications in a hierarchized library, to edit them through a special purpose editor (with a graphical interface), to compile them and to debug them. A symbolic evaluator and theorem proving tools completes ASSPEGIQUE into an environment suitable for rapid prototyping.

I - INTRODUCTION

It is generally agreed on the fact that algebraic specifications provide a powerful tool for writing hierarchical, modular and implementation-independent specifications. Moreover, algebraic specifications are especially suitable for rapid prototyping and are an appropriate framework for verification and validation tool development.

However, some problems have been identified when specifying realistic software using algebraic data types [BBGGG 84]. These problems are the design and management of *large specifications*, *error handling* and *error recovery* policy specification, and the lack of *computer environment* and *tools* supporting the specification stage.

The size of a specification clearly varies in accordance with the complexity of the system being specified. Therefore, specifications of large software systems cannot be managed as a whole. It is necessary to split them into smaller, hierarchized elementary units. Besides, a better modularity promotes the reusability of existing specification parts. Consequently, the design and management of large specifications require tools to structure and modularize the specifications, while the problem of the reusability and integration of existing specification parts must also be addressed. Obviously, structuration, modularization and reusability issues must be taken into account from the first stage of design of the *specification language*. Therefore, specification-building primitives as well as visibility handling primitives should be included into the specification language.

A classical difficulty in the development of large systems is that the error handling specification and the error recovery policy is done too late, very often after the specification of the normal behaviour of the system is completed. This results in expensive modifications of early design decisions. Moreover, the exception handling part of the system is often the less carefully specified. A reason of this sorry state of affair may be that very few methodological and linguistic tools are available to specify and develop software with exception handling. The programming languages which are currently in use in industrial contexts do not provide specific features for raising and handling exceptions. Fortunately, new programming languages, such as Ada [DOD 83, BGG 84], will provide such tools. It is therefore necessary to complete the algebraic specification framework in order to be able to specify error cases and error recovery.

Specific tools must also be provided that support the use of algebraic specifications. First of all, it is now widely agreed on the fact that specification languages and methods without supporting tools are not practicable. Secondly, it is especially important that specific tools with user-friendly interfaces are designed in order to bridge the gap between underlying mathematical formalisms and the user. Such tools should at least comprise intelligent (syntax directed) editors and data base facilities. It is also stressed that graphic interfaces are particularly well-suited for these purposes.

In the remaining of this paper we first give a description of PLUSS-E, a specification language with exception handling and error recovery features, we then describe the design and organization of the ASSPEGIQUE specification environment.

II - THE SPECIFICATION LANGUAGE PLUSS-E

The aim of the family of specification languages **PLUSS** is to provide a tool to express structured algebraic data type specifications. The specification languages of the PLUSS family are based upon a set of specification-building operations derived from the primitive operators suggested by *Martin Wirsing* in **ASL** [WIR 82, S&W 83, WIR 83]. The original design of PLUSS was made by *Marie-Claude Gaudel* [GAU 83].

Roughly speaking, the semantics of the PLUSS specification languages is parameterized by the class of algebras taken into account. More precisely, the semantics of each specification language of the PLUSS family follows some basic, fixed rules in what concerns the specification-building primitives but depends on the class of algebras that are allowed for this specific language. For instance, PLUSS-P will denote the specification language where ***partial algebras*** are chosen as models [BW 82], while PLUSS-0 denotes the specification language where the standard, usual algebras are chosen as models.

Here, we describe **PLUSS-E**, a specification language where exception handling and error recovery can be specified with a precise and formal semantics. PLUSS-E is based upon two formal approaches, the specification-building primitives of ASL and the concept of **E,R-algebras** which allows all forms of error handling (error introduction, error propagation and error recovery) [BID 84].

II.1 - E,R-algebras

Since 1976 [ADJ 76], the classical approach to algebraic data types has been shown to be incompatible with the use of operations that return error messages for some values of their arguments. In this section we describe a new formalism where all forms of error handling can be specified. Our formalism is very closed to the *error-algebras* introduced by Goguen [GOG 77] or to the work described in [GDLE 83], that is, the carrier sets of the algebras are split into okay values and error values. However, we have shown how an implicit error propagation rule may be encoded into the models without losing the possibility of error recovery. Thus all the axioms necessary to specify error propagation may be avoided, and the specifications remain well-structured and easily understandable.

The algebraic specification of error cases, error propagation and error recovery is a difficult problem [ADJ 76, GOG 77, GOG 78, PLA 82, B&W 82, EHR 83]. Our claim is that neither exception cases nor error recovery cases should be specified by means of equations, but rather by means of *declarations*. The axioms of a specification will be divided into four parts:
- Declarations of exception cases and of recovery cases. Declarations are just terms or positive conditional terms (i.e. terms conditioned by equations).
- Ok-axioms.
- Error-axioms.
Ok-axioms or error-axioms are just equations or positive conditional equations.

Thus some terms will be declared to be okay, others will be declared to be erroneous. Ok-axioms and error-axioms will be used to identify ok-values and error-values respectively, no more. This will lead to more structured specifications, since the specification of the error policy (error introduction and error recovery) will be made apart from the axioms. Moreover, our framework will implement the following natural propagation rule:

errors propagate unless their recoveries are specified.

In order to allow a careful recovery policy and the use of non-strict functions, we shall use three distinct kinds of variables: ordinary variables may range over the whole carrier set, ok-variables may only range over the ok-part of the corresponding carrier set, error-variables may only range over the error part of the corresponding carrier set. As a syntactical convenience, ok-variables will always be suffixed by "+", while error-variables will always be suffixed by "-" (e.g. x+, y-, ...).

The necessary underlying theoretical material is described in [BID 84], as well as the study of sufficient conditions for initial models. Here we focus on the feasability of such specifications and their impact. As a first example of a PLUSS-E specification we describe a stack of integers which will allow one underflow, no more:

```
SPEC : STACK
      WITH : INTEGER
      SORTS : Stack

OPERATIONS :
      empty :                         -> Stack
      underflow :                     -> Stack
      crash :                         -> Stack
      push :         Integer Stack    -> Stack
      pop :          Stack            -> Stack
      top :          Stack            -> Integer
```

```
VARIABLES :
        x :   Integer
        p :   Stack
EXCEPTION CASES :
        e1 :   underflow
        e2 :   crash
        e3 :   pop empty
        e4 :   top empty
RECOVERY CASES :
        r1 :   push x+ underflow
OK-AXIOMS :
        ok1 :   pop (push x p) = p
        ok2 :   top (push x p) = x
        ok3 :   push x underflow = push x empty
ERROR-AXIOMS :
        err1 :   pop empty = underflow
        err2 :   pop underflow = crash
        err3 :   push x- p = crash
END STACK.
```

In this stack, we shall have an infinite number of error elements, with two specific values: *underflow*, which will be obtained (as the result) when popping the empty stack; and *crash*, which will be obtained when popping *underflow*. Stack terms obtained from the *crash* stack are definitively erroneous. *Underflow* is an erroneous stack, but one can recover from this state by pushing an okay integer onto it. In all cases pushing an erroneous integer onto a stack leads to the *crash* stack.

Note also that nothing more is required than "*top empty* is an exception case"; however, if one wants to identify *top empty* with an erroneous integer, say *bottom*, an error-equation "*top empty = bottom*" may be added. Furthermore, if one wants to identify all erroneous integers with *bottom*, this can be achieved by adding the following error-equation: "*x = bottom*".

In the same way, the equation ok3 is not absolutely necessary; however, in our case we do not want to just specify that pushing an okay integer onto the *underflow* stack is a recovery case, but also that the stack obtained is equal to pushing the same element onto the *empty* stack. One explication is also needed for the error-equation err3: note that we have not (explicitely) specified that *push x- p* is an error value; this is simply a consequence of the natural error propagation rule, since *x-* denotes an error value.

II.2 - Basic specifications

In PLUSS-E, a basic specification is a *signature* together with *axioms*. Axioms are preceded by the declaration of the *variables* they use.

A *signature* begins with the key-word SORTS followed by a list of sorts. By default an empty list is equivalent to the declaration of a unique sort, which is equal to the specification name. If no new sort is required, the key-word SORT may be omitted or be replaced by the key-word NO-NEW-SORT.

The second part of the *signature* is a list of operation names together with their arity. This list begins with the key-word OPERATIONS. For instance:

_[_] : Array Index -> Elem

As it appears in the above example, mixfix operators with a syntax à la OBJ [GOG 83] are allowed, and the underscore is used to indicate where the operation arguments should be placed. Thus the above operation may be used to access an array element with the usual notation t[i]. Underscores may be omitted if the operation will be used in the standard prefix order. Overloading of operation names and coercions are also allowed, for instance:

_ : Integer -> Real

The declaration of the *variables* used in the axiom part is preceded by the key-word VARIABLES. All the variables are implicitely universally quantified. As well as for operation names, variable names can be overloaded; this is especially useful for dealing with structured specifications. According to the syntactical convenience described in Section II.1, the declaration of a variable x implicitely contains the declaration of x+ and x-.

Declarations and *axioms* are named, and positive conditional equations (or declarations) are allowed. They are preceded by the key-words EXCEPTION CASES, RECOVERY CASES, OK-AXIOMS and ERROR-AXIOMS. An example of one of the conditional axioms that define *less or equal* on the integers is given below:

LE5 : 0 LE x = true => 0 LE s x = true

Since the = sign is used to connect both terms of an equation, it can not be used as a name for the equality operation. We suggest to use _*is* _ to this end, as well as _*isnot* _ for the inequality operation.

As a last remark on basic specifications, one should note that an implicit **reachable** is embedded in such a specification. That means that only finitely generated models are taken into consideration.

II.3 - Specification construction tools

The simplest feature to build specifications from simpler ones is the **sum**, which is denoted by +. By definition, the signature of SPEC + SPEC' is the union of the signatures of SPEC and SPEC'. There is no implicit renaming as in *CLEAR* [B&G 79]. So it is possible to share subspecifications without duplicating them: for instance, if there is a sort Bool in SPEC and in SPEC', there is only a sort Bool in SPEC + SPEC'. The same rule applies for operation names. The meaning of such a specification is a class of algebras w.r.t. this signature. The algebras of this class must be models of SPEC (resp. SPEC') when they are restricted to the signature of SPEC (resp. SPEC'). Note that the concept of restriction must be suitably refined w.r.t. E,R-algebras.

A by-product of the sum is the **enrichment**, denoted by WITH, which allows to add new sorts and/or new operations and/or new axioms to a specification. An example of this construct was given in the one-error-tolerant stack. Since specification names may also be overloaded, one can specify the file where the specification to be enriched has to be found. This can be done using the FROM option, e.g.:

```
WITH : BOOL        FROM : Std-BOOL
                or
WITH : BOOL        FROM : My-BOOL
```

Since there is no implicit renaming, **explicit renaming** is needed in the language. The syntax is straightforward:

RENAMING sort1 INTO sort2; op1 INTO op2; ... END

Sorts or operation names may be **forgotten** by writing *"name INTO"*.

A specification may be **parameterized** by another specification. The signature and the axioms of the formal parameter express the properties which are required for the argument. An argument is a couple made of a specification and a signature morphism. The signature morphism is called the *fitting morphism*. It sends sorts and operation names of the formal parameter into the relevant sorts and operations of the argument. The meaning of the application of an argument <ARG, m> to a parameterized specification SPEC(X) depends on the correctness of ARG with respect to X. The models of ARG, restricted to the signature of X w.r.t. m, must be models of X. The signature of the resulting specification is the union of the SPEC signature, where sorts and operation names coming from X are renamed by m, and of the ARG signature (i.e. the X signature disappears). The models are those of the specification w.r.t. the resulting signature.

An example of a parameterized specification, ARRAY (ELEM, INDEX), is given below. The ELEM and INDEX specifications are formal parameters. The ELEM specification does not require anything:

```
PAR : ELEM
      SORTS : Elem
END ELEM.
```

The INDEX specification is slightly richer, since the argument must provide a maximum and a minimum index, and a total ordering on the indexes. It is not given here for lack of space.

```
PROC : ARRAY (ELEM, INDEX)
      SORTS : Array

OPERATIONS :
      init:          Index Index          -> Array
      lwb:           Array                -> Index
      upb:           Array                -> Index
      _[ _]:= _:     Array Index Elem     -> Array
      _[ _]:         Array Index          -> Elem

VARIABLES :
      t :      Array
      i, j :   Index
      v, v' :  Elem

EXCEPTION CASES :
      illegal-access :   (i < lwb t) or (i > upb t) = true    => t[i]
      illegal-modif :    (i < lwb t) or (i > upb t) = true    => t[i]:=v
      illegal-init :     i > j = true                         => init i j
```

```
OK-AXIOMS :
     bound1 :                         lwb (init i j)        = i
     bound2 :                         upb (init i j)        = j
     bound3 :                         lwb (t[i]:=v)         = lwb t
     bound4 :                         upb (t[i]:=v)         = upb t
     access1 :   i is j = true     => (t[i]:=v) [j]         = v
     access2 :   i isnot j = true  => (t[i]:=v) [j]         = t[j]
     modif1 :    i is j = true     => (t[i]:=v) [j]:=v'     = t[i]:=v'
     modif2 :    i isnot j = true  => (t[i]:=v) [j]:=v'     = (t[j]:=v') [i]:=v
END ARRAY.
```

Note that there is an implicit enrichment of the formal parameters. Thus no WITH is required in the above specification. Furthermore, there is **no implicit reachable** embedded in the specification of the formal parameters, since (finitely generated) models of actual parameters may not be finitely generated models of the formal parameters.

An example of a specification built by instantiation of ARRAY is given below:

```
SPEC : STRING
     ARRAY (ELEM => CHAR, INDEX => INT25)
     RENAMING   Array   INTO String
                init    INTO emptystring
END STRING.
```

This overview of PLUSS-E is of course very incomplete. For instance, other features of this specification language deal with visibility rules and allow the user to hide some private specifications and/or sorts and/or operations.

III - THE ASSPEGIQUE ENVIRONMENT

In the previous section, we have described how exception handling and structuration features can be embedded into an algebraic specification language. Our claim is that a specification language should not only be supported by some specification method, but also by specific tools integrated into a specification environment. Parts of such environments have already been designed or described, such as OBJ, AFFIRM, the CIP Project and the LARCH Project. The main characteristics of these environments can be roughly outlined as follows:
- OBJ [GOG 81, GOG 83] has been developed in order to experiment theoretical hypothesis and specification combining tools.
- AFFIRM [AFF 81] is oriented towards proof purposes. No generic tool is provided to combine specifications. Full-sized examples were developed with this system.
- The CIP Project [CIP 81] attempts to consider in a uniform way specifications and programs.
- The LARCH Project [G&H 83] is developing tools and techniques intended to aid in the productive use of formal specifications of systems containing computer programs.

Our goal when designing the ASSPEGIQUE specification environment was to try to integrate all these characteristics. The main aspect of our specification environment is the high degree to which it allows specifications to be **modularized**. The user of ASSPEGIQUE deals with *elementary* specifications which correspond, so to speak, to *types of interest*. A major role of the specification language is to assemble these elementary specifications into more complex ones.

Modularity imposes constraints that led us to introduce sophisticated mechanisms in the specification environment. For instance, it is useful to be able to specify a type of interest without having previously specified all the *predefined* types required by this type of interest. We did in fact find ourselves obliged to abandon a wholly top-down approach in the development of elementary specifications : such an approach leaves to last those checks that the system must carry out if it is of any real value to the user. Leaving those checks to the end however means that errors may be discovered far too late. We were also obliged to abandon a wholly bottom-up approach, because such an approach does not correspond to a natural way of specifying complex data structures : the most primitive specifications are generally given last, and not first.

III.1 - Hierarchical relations within the specification library

The high degree of modularization of the library and the impossibility to follow an approach which is either strictly bottom-up or strictly top-down, mean that we have to provide the library with ordering relations : on the one hand, such relations allow to reconstitute, whenever necessary, all the *required types* (i.e. those assumed to be predefined) for a given type of interest; on the other hand, they allow to manage the consequences of a modification made on any of the elementary specifications in the library.

Two ordering relations are defined on the specification library ; before defining them formally, we shall explain their use with a simple example.

$$\begin{tikzcd}
 & \text{Stack-of-integers} \arrow[dl] \arrow[dd] \arrow[dr, Rightarrow] & & \\
\text{Integer} & & \text{Stack(X)} \arrow[dl] \arrow[dr] & \\
 & \text{Bool} & & \text{X}
\end{tikzcd}$$

$\longrightarrow$ directly requires
$\Longrightarrow$ instance of

Graph of ordering relations associated with Stack-of-integers

Operations which may appear amongst the axioms of Stack-of-integers are operations on Stack(X) (instantiated), but also operations on Bool and Integer; consequently checks to be carried out on the axioms of Stack-of-integers (e.g. that all terms are well-formed) cannot be carried out before specifications Integer and Bool have been introduced into the library (since nothing prevents us from defining Integer or Stack(X) before defining Bool) : Stack-of-integers requires {Integer, Bool}.
Moreover any modification to X or Stack(X) must lead to reconsideration of Stack-of-integers : after a modification Integer may no longer be a suitable parameter.

More precisely, the ordering relations provided within the library are defined as follows :
1. The ordering relation *requires* is defined as being the transitive and reflexive closure of the relation *directly requires*, defined as follows :

a specification S2 *directly requires* a specification S1 if and only if S2 is built as an enrichment of S1 (S1 is on the list of specifications enriched by S2) or if S2 is an instance of a parameterized specification which directly requires S1 (e.g. in the example above, Stack-of-integers is an instance of Stack(X), which directly requires Bool; it is self-evident that the instantiated parameter is excluded from the definition above : Stack(X) directly requires X, but Stack-of-integers does not).

2. The ordering relation *depends on* is defined as being the transitive and reflexive closure of *directly requires or is an instance of*.

N.B. : As mentioned above, the relation *requires* will in particular allow computation of the set of all the operations that may appear in an axiom of the specification considered, while the relation *depends on* will allow to propagate modification consequences in the library.

III.2 - Overall organization of ASSPEGIQUE

The tools available in the ASSPEGIQUE environment include : a **special purpose editor**, **modification tools**, a **compiler**, a **debugger**, a **symbolic evaluator** and **theorem proving tools.** They are available to the user through a **user interface** and access the specification library through the **hierarchical library management tool** (see figure on next page).

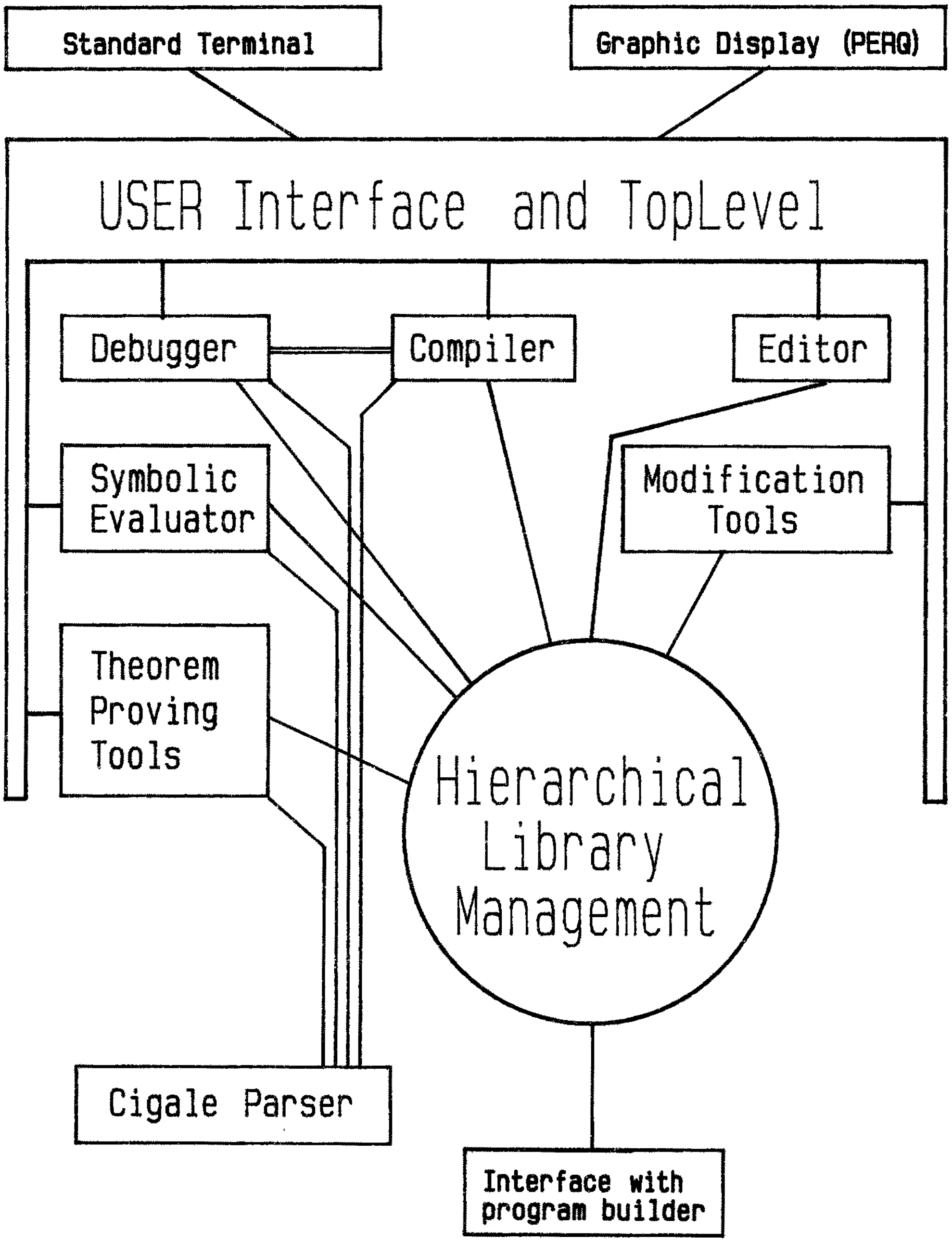
Particular attention has been paid to the **interaction with the user** : all the tools are interfaced in a uniform way and make full use of full-screen multiwindow display and graphic facilities to provide a user-friendly interaction ; among others, this includes the use of on-screen and pop-up menus, and on-line help and documentation facilities, detailed according to the degree of expertise of the user.

The role of the **hierarchical library management tool** is to maintain the library coherence w.r.t. the ordering relations between the specifications ; in particular, the management tool updates the library information file.

The **specification editor** (cf. III.3) is syntax-directed which does not mean that the user has to deal with the internal representation of the specification : the *concrete views* available to the user are a *text representation* of the specification and a *graphic representation* of its signature. Modifications and creations are performed by the user at the level of these concrete views, the corresponding internal representation being accordingly updated.

The **specification compiler** plays two roles :

1. It computes the *grammar* associated with the specification; this grammar allows to analyse the axioms defined in the specification, to verify that they are syntactically correct and to resolve overloading conflicts. To achieve these purposes, the specification compiler uses the grammar generator and the parser described in [VOI 84].
The grammar associated with a specification S is defined as the union of grammars of all specifications required by S and of operations defined within S. Consequently, the compilation of a specification cannot take place before all specifications on which it depends have been introduced. The compilation of a specification may moreover require that specifications which are required but have not yet been compiled (or have been changed since their last compilation) are compiled first.
2. The compiler produces an internal form of the specification which will be directly usable by software accessing the library : symbolic evaluator, theorem prover, program construction assistance tool [BGG 83].


Standard Terminal
Graphic Display (PERQ)
USER Interface and TopLevel
Debugger
Compiler
Editor
Symbolic Evaluator
Modification Tools
Theorem Proving Tools
Hierarchical Library Management
Cigale Parser
Interface with program builder

The internal form associated with a specification is made up of LISP property lists : this type of internal form is especially flexible, and has shown itself very convenient for handling problems raised when interfacing ASSPEGIQUE with external tools.

The **specification debugger** is automatically loaded by the compiler when errors are detected ; it allows interactive debugging of the specification. In particular, and this is a contrast with the editor, the debugger is not loaded unless all required specifications do in fact exist (and have been compiled). The debugger consequently allows to debug axioms interactively, which is impossible at the general editing level.

The **symbolic evaluator** computes the canonical form of any term w.r.t. the axioms. The symbolic evaluator takes into account conditional axioms and parameterization [KAP 83].
The **theorem proving tools** use the techniques described in [BID 82].

Finally, the specification environment is provided with tools whose aim is to make operations easier and to maximize the possibilities for **re-using** specifications. Consequently suitable tools allow to copy a specification, to enrich it by adding new operations or axioms, or to rename an operation or an axiom, etc.

III.3 - The ASSPEGIQUE syntax-directed full-screen editor

As outlined before, the originality of the editor is that, while offering all the flexibility in use of a full-screen editor, it establishes links between the external concrete views (text, graphics) and the internal representation, making it a syntax editor [EDS 83]. Any movement within the text or the graphic representation of the description is also a movement within the corresponding tree, and any modification is taken into account at both levels by means of a validation mechanism. The part of the editor devoted to the specification text was developed as an extension of WINNIE [AMA 83], a full-screen multi-window editor. The graphical part of the editor was derived from a graphical interactive editor for Petri-nets, PETRIPOTE [BEA 83].

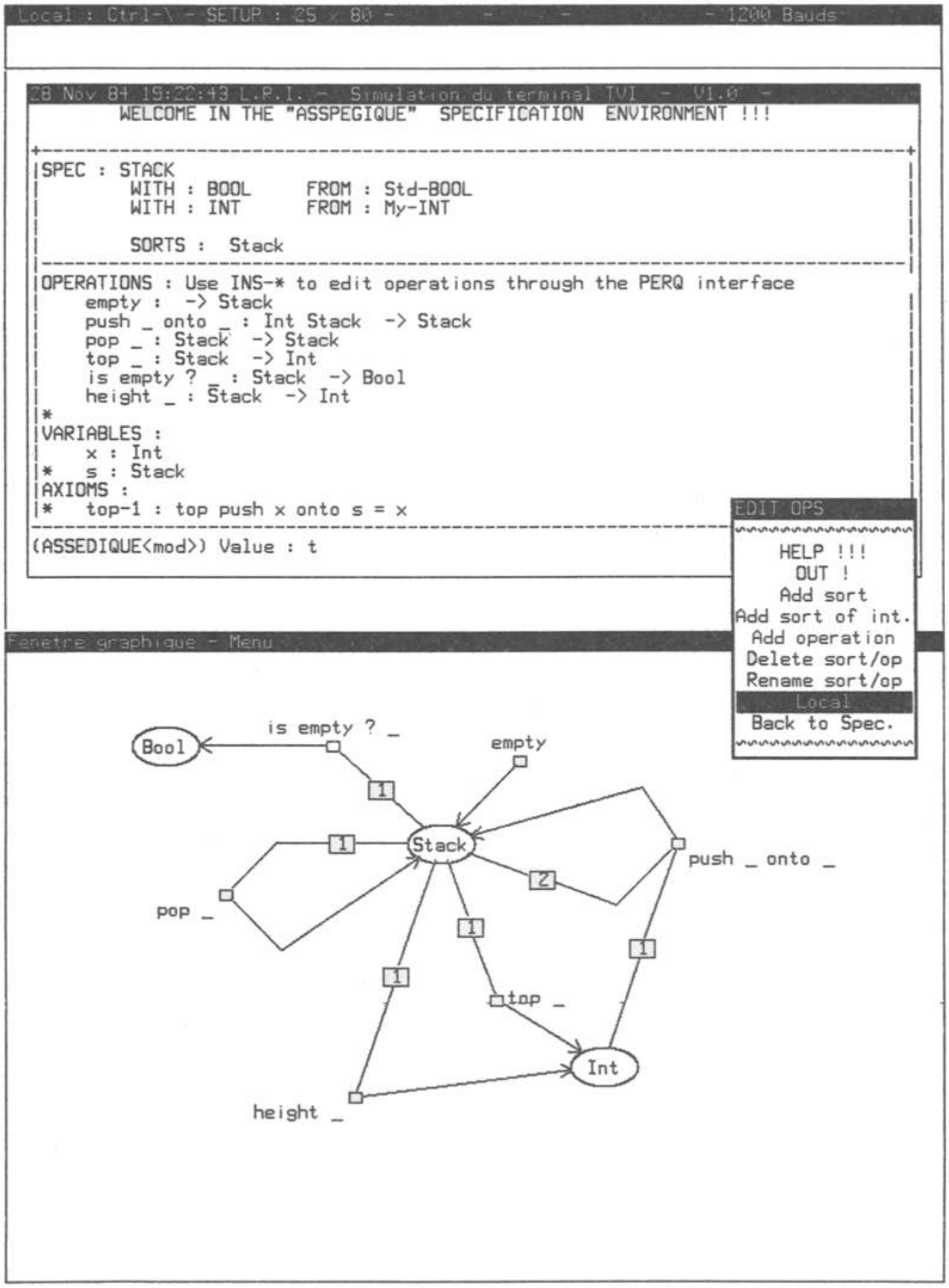
The editor displays a template which depends on the construction primitive (enrichment, formal parameter, parameterized type, etc...), and on the *style* of specification (basic, with error handling, etc...). For instance, the figure on next page shows how the screen looks like at some stage of the edition of a specification *enriching* other ones, written in a ***basic style***.

The template displayed may be broken down into :

1. A heading part which includes the name of the type (SPEC clause), the names of the required types (WITH clauses) and the names of the files containing their descriptions (FROM clauses), as well as the names of sorts involved in the type description (SORTS clause); the user may create as many WITH FROM clauses as there are required types.

2. An OPERATIONS part, in which the name and syntax of each operator is specified (note that the parser and incremental grammar generator [VOI 84] allow the user to indicate where the operation arguments should be placed).

3. A VARIABLES part : name and type of variables.



```
Local : Ctrl-\ - SETUP : 25 x 80 -          -       -          - 1200 Bauds

28 Nov 84 15:22:43 L.R.I. -  Simulation du terminal TVI  -  V1.0  -
        WELCOME IN THE "ASSPEGIQUE"  SPECIFICATION  ENVIRONMENT !!!

+--------------------------------------------------------------------------+
|SPEC : STACK                                                              |
|        WITH : BOOL      FROM : Std-BOOL                                  |
|        WITH : INT       FROM : My-INT                                    |
|                                                                          |
|        SORTS :  Stack                                                    |
|--------------------------------------------------------------------------|
|OPERATIONS : Use INS-* to edit operations through the PERQ interface
|    empty :  -> Stack                                                     |
|    push _ onto _ : Int Stack  -> Stack                                   |
|    pop _ : Stack  -> Stack                                               |
|    top _ : Stack  -> Int                                                 |
|    is empty ? _ : Stack  -> Bool                                         |
|    height _ : Stack  -> Int                                              |
|*                                                                         |
|VARIABLES :                                                               |
|    x : Int                                                               |
|*   s : Stack                                                             |
|AXIOMS :                                                                  |
|*   top-1 : top push x onto s = x                                         |
----------------------------------------------------------------------------
(ASSEDIQUE<mod>) Value : t

Fenetre graphique - Menu
```

4. The AXIOMS part (which would be divided into four sub-parts, if the specification were written in the *error handling style*.)

Any declaration (name of type, name and syntax of operation, etc...) is called an entity. A star indicates that the following entity has not been validated yet. Validation consists in checking the syntax of an entity declaration, and in completing the corresponding sub-tree of the internal representation.

In what concerns the signature (sorts and operations), the user may either type in directly the text in the text view or use the PERQ graphical interface. In this case, (s)he just needs to select the appropriate command in the pop-up menu and draw, say, the operation, by pointing at the domains and codomain sorts (the operation arrows are then drawn automatically by the system).

IV - CONCLUSION

In this paper, we have shown how to systematically cope with structuration and exception handling at the specification level by providing an appropriate specification language; we also described a specification environment that supports such a language and therefore its practical use. The tools integrated in ASSPEGIQUE range from a high-level syntax-directed editor to a symbolic evaluator and theorem proving tools; therefore, PLUSS-E and ASSPEGIQUE are especially well-suited for rapid prototyping purposes [CHO 85]. Industrial sized experimentations on ASSPEGIQUE are currently under development, and will provide a firm basis to further versions.

A kernel system of ASSPEGIQUE consisting in the top-level and user-interface, the specification editor together with its graphic interface, a specification compiler, a symbolic evaluator, the special purpose grammar generator and the parser, theorem proving tools has been implemented on VAX-UNIX and PERQ-POS. This system has been demonstrated at the 7th International Conference on Software Engineering, Orlando, USA and at the 2nd AFCET Conference on Software Engineering, Nice, France.

V - ACKNOWLEDGEMENTS

This work is partially supported by A.D.I. Contract No 639 and the C.N.R.S. (Greco de Programmation).

Special thanks are due to *Marie-Claude Gaudel* for many helpful suggestions and discussions in designing the PLUSS-E language. We also thank *Frederic Voisin*, *Stephane Kaplan*, *Marianne Choquer*, *Michel Beaudouin-Lafon* who respectively made the parser, the symbolic evaluator, the theorem proving tools and the graphic interface.

VI - REFERENCES

[ADJ 76] Goguen J., Thatcher J., Wagner E., "An Initial Algebra approach to the specification, correctness, and implementation of abstract data types" in Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978 (also IBM Report RC 6487, October 1976).

[AFF 81] Gerhart S.L., "AFFIRM Reference Manual", UCS-Report (Marina del Rey), 1981.

[AMA 83] Amar P., "Winnie : un éditeur de textes multifenêtres extensible", Actes des Journées BIGRE (Le Cap d'Agde), 1983.

[BBGGG 84] Bidoit M., Biebow B., Gaudel M.-C., Gresse C., Guiho G., "Exception handling : formal specification and systematic program specification", Proc. 7th I.C.S.E., Orlando, USA, 1984.

[BEA 83] Beaudouin-Lafon M., "Petripote: a graphic system for Petri-Nets design and simulation" Proc. of 4th European Workshop on Applications and Theory of Petri Nets, Toulouse, France, 1983.

[BID 82] Bidoit M., "Proofs by induction in "fairly" specified equational theories" Proc. 6th German Workshop on Artificial Intelligence, Bad Honnef, Germany, Springer-Verlag IFB 58, 1982.

[BID 84] Bidoit M., "Algebraic specification of exception handling and error recovery by means of declarations and equations", Proc. 11th ICALP, Antwerp, 1984.

[BCK 84] Bidoit M., Choppy C., Kaplan S., "ASSPEGIQUE : un environnement de spécification algébrique", Proc. 2nd AFCET Software Engineering Conference, Nice, 1984, pp357-371

[BGG 83] Bidoit M., Gresse C., Guiho G., "CATY : Un système d'aide au développement de programmes", Actes des Journées BIGRE 83 (Le Cap d'Agde), 1983.

[BGG 84] Bidoit M., Gaudel M.-C., Guiho G. "Towards a systematic and safe programming of exception handling in ADA" Proc. of Ada-Europe/Ada TEC Conf., Brussels, June 1984.

[B&W 82] Broy M., Wirsing M., "Partial Abstract Data Types" Acta Informatica, Vol.18-1, Nov 1982.

[B&G 79] Burstall R., Goguen J., "The semantics of CLEAR, a specification language", in Abstract Software Specifications, D. Bjorner Ed., LNCS 86, Springer-Verlag, 1979.

[CIP 81] CIP language group "Report on a wide spectrum language for program specification and development", Rapport TUM-I8104 (Munich), 1981.

[CHO 85] Choppy C., "Tools and techniques for building rapid prototypes", AFCET Workshop on "Prototypage, Maquettage et Génie Logiciel", Lyon, January 1985.

[DOD 83] "The programming language ADA - Reference Manual" United States Department of Defense, January 1983.

[EDS 83] "Les éditeurs dirigés par la syntaxe", Journées d'Aussois - INRIA Ed. (Rocquencourt - France), 1983.

[GAU 83] Gaudel M.-C., "Proposition pour un Langage d'Utilisation de Spécifications Structurées : PLUSS", C.G.E. Research Report, 1983.

[GDLE 83] Gogolla M., Drosten K., Lipeck U., Ehrich H., "Algebraic and operational semantics of specifications allowing exceptions and errors" Proc. 6th GI-Conference on Theoretical Computer Science, LNCS 145, 1983, Springer-Verlag.

[GOG 77] Goguen J.A., "Abstract errors for abstract data types" in Formal Description of Programming Concepts, E.J. NEUHOLD Ed., North Holland, New York 1977.

[GOG 78] Goguen J.A., "Exception and Error Sorts, Coercion and Overloading Operators" S.R.I. Research Report, 1978.

[GOG 81] Goguen J.A., Parsaye-Ghomi K., "Algebraic denotational semantics using parameterized modules", Tech. Report CSL-119, SRI International, UCLA, 1981.

[GOG 83] Goguen J.A., "Parameterized Programming", Proc. Workshop on Reusabibility in Programming, Stratford CT, USA, 1983.

[G&H 83] Guttag J.V., Horning J.J., "An introduction to the LARCH shared language", Proc. IFIP 83, REA. Mason ed., North Holland Publishing Company, 1983.

[KAP 83] Kaplan S., "Un langage de spécification de types abstraits algébriques", Thèse de 3ème cycle, LRI (Orsay - France), 1983.

[PLA 82] Plaisted D., "An initial algebra semantics for error presentations" Unpublished Draft, 1982.

[S&W 83] Sanella D., Wirsing M., "A Kernel Language for Algebraic Specification and Implementation", to appear in Int. Conf. on Foundations of Computing Theory, Bergholm, Sweden, 1983.

[VOI 84] Voisin F., "CIGALE : un outil de construction incrémental de grammaire et d'analyse d'expression", Thèse de 3ème cycle, Orsay (France), 1984.

[WIR 82] Wirsing M., "Structured algebraic specifications", Proc. AFCET Symp. on Mathematics for Computer Science, Paris, France, 1982.

[WIR 83] Wirsing M., "Structured algebraic specifications : A kernel language", PhD. Thesis, Munchen, Germany, 1983.