Interchange Format for Inter-Operability of Tools and Translation

The Salsa and Asspegique$^+$/LP Experience *

Michel Bidoit,[1] Christine Choppy [2,3] and Frédéric Voisin [3]

[1] LIENS, C.N.R.S. U.R.A. 1327 & Ecole Normale Supérieure
45 Rue d'Ulm, F–75230 Paris Cedex 05, France
[2] IRIN, Université de Nantes & Ecole Centrale
F–44072 Nantes Cedex 03, France
[3] LRI, C.N.R.S. U.R.A. 410 & Université de Paris-Sud
Bât. 490, F–91405 Orsay Cedex, France

Abstract. This paper presents a technical approach for solving the problem of inter-operability between the different existing tools and languages for algebraic specifications. We discuss some semantic compatibility criteria required to ensure a minimal level of inter-operability, we describe a proposal for a specification interchange format, and we explain the mechanisms needed to solve the issues raised by user interaction. The technical generic solutions described in this paper were implemented and experimented in the framework of the Salsa project and when building an interface between Asspegique$^+$ and LP.

1 Introduction

In order to make use of algebraic specifications the essential ingredients are a specification language and an environment where tools to design, verify and use specifications are integrated. Algebraic specifications have been investigated for over twenty years now, one of the effects being that various algebraic specification languages were designed and experimented, and that several tools and/or environments for algebraic specifications were developed (see e.g. [7, 29] for references to both algebraic specification languages and tools). The fact that many languages and tools have been developed can be considered to be beneficial, in so far as it allows designers and users to experiment with various features. However, a detrimental counterpart of this proliferation is the incompatibility of these various languages and tools. In practice, when one wants to apply some tool on some algebraic specification, it is only possible to consider the specific tools that were designed to support the algebraic specification language used, even if some other tools would seem more appropriate or more powerful. Reuse

* Work partially supported by the ESPRIT Basic Research Working Group 6112 COMPASS-II, by a CNRS/NSF grant, and by the CNRS GDR-PRC "Programmation".

or exchange of specification libraries is also in most cases an intractable task.

Developing tools and environments requires a substantial amount of work, and new tools are made available each year, so it is not possible in practice for a given environment to integrate, for each kind of tools, the best available one. Our aim is to provide a reasonably easy way to access existing tools, in order to be able to apply them on a given specification, in particular when the tools considered were not initially designed to support the algebraic specification language used to express the specification. There is a very strong constraint in our problem which is that we do not want to recode the tools to be used, and if we need to modify them for our purpose, the implied coding task should be minimal. The idea here is to be able to take advantage of existing tools without having to develop them anew. Clearly, this issue is quite different from the one of developing new tools that would be designed from the beginning in such a way that they will be compatible with any algebraic specification language (or at least with a broad family of algebraic specification languages). In this paper, we discuss both semantical and practical issues raised by our goal, and then we present the pragmatic solution that we have designed to solve the practical side of the problem.

We did our first experiment in this matter more than ten years ago, when we wanted to interface Reve [23, 16] with our Asspegique system [5, 10, 12]. The specification language supported by Asspegique was a subset of the Pluss language [6, 4, 19], including basic specifications, enrichment and sum of specifications.[4] The axioms allowed were conditional equations. The Reve system supported only flat equational specifications. Our motivation for designing an interface between Reve and Asspegique was that Reve provided an automatic ordering computation to orient equations into finitely terminating rewrite rules, Knuth-Bendix completion, and inductionless induction, features that were not available in Asspegique. The rewriting tool in Asspegique assumed that the equations must be oriented from left to right, and provided graphic tracing tools and mixed rewriting capabilities [22] (that were not available in Reve). The theorem proving tool of Asspegique relied on explicit induction techniques. Hence, an interface between Reve and Asspegique allowed the user to check that the naive left to right orientation of equations into rewrite rules led to a finitely terminating rewrite system, or to use the finitely terminating rewrite system computed by Reve as input for the Asspegique rewriting tool, or even to select the most appropriate proof technique (inductionless induction versus explicit induction) depending on the specific specification context. Since Reve supported equations only, the interface was not usable on specifications with conditional equations, but only on Asspegique specifications restricted to equational axioms. Another problem was that Reve had syntactic and lexical conventions that were quite different from those of Asspegique: apart from some special

[4] More features (genericity, renaming, etc.) are available in the latest version called Asspegique$^+$ [12, 25].

cases (+, *, <, etc.), operation names are prefixed in REVE, while they can be mixfixed in ASSPEGIQUE; in REVE, any name beginning with a letter between "a" and "t" is a function name, while a name beginning with a letter between "u" and "z" is a variable name, and symbols like +, *, < are necessarily binary infix symbols. This led us to transform the operation and variable names of the ASSPEGIQUE specification so as to generate an appropriate unambiguous REVE input (cf. [10]). For instance, the axiom "`fact s x = (s x) * fact x`" was translated into "`f_fact(f_s(v_x)) == f_Rmult(f_s(v_x), f_fact(v_x))`". While ASSPEGIQUE's users were able to use REVE without having to retype from scratch their specifications, the translation was not very legible, and it was not very easy for the user to type a REVE command involving a term, or to read the results output by REVE.[5] So, at that time this interface problem between ASSPEGIQUE and REVE was solved in a simple and not fully satisfactory way. One of the reasons for this was that we had designed and implemented the interface with the idea that it should/could not imply any modification of the REVE source code; another was that we strongly underestimated the difficulty of the issues raised by such interfaces (but it was our first experiment). Indeed in this paper we will present a more elaborate solution that leads to much better results from a user point of view, but that requires in fact some work from both environments' developers. We will show that this is mandatory for the kind of interaction needed for controlling e.g. a proof assistant like LP [17, 18].

This paper is organized as follows. In the next section we discuss the various issues raised by inter-operability of tools and exchange of algebraic specifications from a general viewpoint. In Section 3 we describe the "specification interchange format" we have designed and experimented in the framework of the SALSA project [21].[6] In Section 4 we discuss various problems to be solved in order to make the inter-operability of tools as transparent for the user as possible.

2 Issues Raised by Inter-Operability

In this section we discuss the main issues to be solved when trying to achieve a reasonable level of inter-operability between tools and algebraic specification languages. Our aim is to be able to operate some tool, hereafter called TOOL, on some specification SP written in some algebraic specification language ASL[7] (with the implicit assumption that TOOL was not originally designed for the

[5] Only the ASSPEGIQUE specification was automatically translated; for the rest of the interaction, the user had to perform himself the required translations.

[6] The aim of the SALSA project (1991–1993) was to federate the french research teams in the area of algebraic specifications, and in particular to integrate their languages and tools.

[7] The acronym ASL is used here to denote an arbitrary algebraic specification language. We hope there will be no confusion with the kernel ASL language originally designed by Martin Wirsing [28].

ASL language).

Although, as explained in the introduction, this goal is highly desirable, it may seem at first glance far too challenging; in fact, it is true that in some (most?) cases the task is hopeless. However, there are a lot of "simple" cases where the task seems fairly easy (although, as we will see later, it is not). A first simplification arises from the fact that we do not expect to be able to operate TOOL on any specification written in ASL, but only on some of the specifications expressible in ASL (cf. e.g. our example of an interface between REVE and ASSPEGIQUE).

Before discussing more deeply the issues involved by inter-operability, let us first detail some general assumptions on ASL and TOOL. We can assume that the definition of the algebraic specification language ASL is made of:

1. The institution or logical framework used to describe the semantics of ASL (e.g., algebraic signatures plus standard algebras plus many-sorted first-order logic with equality, or order-sorted signatures plus order-sorted algebras plus universally quantified equations, etc.).
2. The so-called "specification-building" primitives of ASL (e.g., enrichment, sum, renaming, parameterization/instantiation, etc.) used to compose specifications and build specification expressions.
3. The (concrete) syntax of the language. For our purpose it is important to note that, besides the rules explaining how to introduce the main pieces of a specification text such as e.g. sorts, operation names and axioms, the syntax includes also special rules w.r.t. which kind of operator names are allowed (e.g., is overloading allowed, can we have infix or mixfix operator names, etc.). Note that the flexibility allowed to name operators has some impact on the way terms used in the axioms can/should be written (e.g. `push(x,p)` versus `push x on p`). In our context it is therefore important to distinguish, in addition to the syntax of the ASL language, the *term language* (induced by the syntactic facilities of the language and the signature of the specification considered) and the *formula language* (induced by the term language and the syntactic aspects of ASL related to formulas).
4. Its "algebraic semantics": usually this semantics associates to each (syntactically correct and well-formed) specification text a signature and a class of models. In general the semantics is compositional w.r.t. the specification-building primitives. Note that some algebraic specification languages have a presentation semantics (cf. e.g. ACT-ONE [15]), or a theory semantics (cf. e.g. CLEAR [8]), but in these cases also we can consider that we are provided with a model semantics [29].

As regards TOOL the situation is less clear. However, if we think of the most used tools, such as e.g. rewrite engines, theorem provers, proof checkers, etc., we may consider as well that we are provided with:

1. Some kind of logical framework, which in general does not need to be an

institution. We are afraid we must stay rather intuitive here, but one may think to e.g. term rewriting systems, conditional term rewriting systems (and all the variants including term rewriting with priority, with strategies, order-sorted term rewriting, etc.), (many-sorted) equational logic, (many-sorted) first-order logic, etc. Typically there is a notion of signature and some concept of "formula" (such as e.g. a rewrite rule) as part of this "logical framework".
2. The concrete syntax of the language recognized by the tool. Typically, the language recognized can be split into the *command language* on the one hand, and the *formula/term* language on the other hand.

Note that most tools do not consider structured sets of "formulas", i.e. in general there is no concept of "specification-building" primitives for TOOL. We may expect, however, that this situation will change in a near future (cf. e.g. [27]).

Given this rather intuitive view of both algebraic specification languages and tools, we may now better understand the issues involved to operate TOOL on the specification SP written in ASL:

1. We need some "compatibility" between the institution underlying the semantics of ASL and the "logical framework" of the TOOL. When this logical framework can be seen as part of an institution itself, then we need some kind of "institution embedding", following e.g. the lines described in [20, 9, 24, 26]. Other kinds of mechanisms may be needed here as well: assume e.g. that ASL is defined using the usual institution of (many-sorted) equational logic, and that TOOL is just a rewrite engine; then we may have to apply first some Knuth-Bendix completion procedure to the set of equations introduced by SP before doing some term rewriting in a truly meaningful way. We may also have to order the axioms of SP in a specific way when TOOL requires the formulas to be introduced in a specific order (as it is often the case for tools based on narrowing or resolution, since these tools may loop or not depending on the order of the formulas...).
Clearly, this issue requires much more work. We need not only theoretical results to relate various institutions and logical frameworks, but also practical means associated to these potential theoretical "embeddings". It would be of great practical relevance to know, for instance, how to apply in a meaningful way a rewrite engine designed for total functions to a specification involving partial operations.
2. A second issue is the so-called "flattening" problem. Since in general the logical framework of the TOOL does not involve any structuring mechanism, we will have to derive from the (a priori) structured specification SP a *presentation*, i.e. a pair made of a signature and a finite set of formulas. It is well-known that in general it is not possible to derive for any specification SP an equivalent presentation (i.e. a presentation which defines the same theory or the same model class as SP). However, in most cases it is more or less easy to derive a presentation that defines a larger (but close enough) class of models than SP, and this may be enough for our purposes [11].

It is important to understand that this "flattening" process can be handled at various levels. First, one can try to perform it inside the institution underlying the semantics of ASL: in this case we will perform the "flattening" first, and then consider the "embedding" from the institution underlying ASL to the logical framework of the TOOL. On the contrary, one can combine "embedding" and "flattening" into a one step "translation". In some cases, it may even be necessary to use an intermediate institution as the target of the "flattening" process, and then consider an "embedding" from this intermediate institution to the logical framework of the TOOL.

We would like to emphasize the fact that the best way to proceed w.r.t. this issue is in general best decided on the basis of the specific TOOL considered. That is, for the same specification language ASL, we may consider distinct ways of combining "embedding" and "flattening" depending on the TOOL we want to use.

Note also that this issue is not really specific to our inter-operability problem. Indeed this issue has already to be solved when designing a tool tailored to the given specification language ASL.[8] Therefore we will not further discuss this problem in what follows.

3. More generally, we need some kind of "semantic compatibility" between the specification SP we consider and the input we will provide to the TOOL. We believe that to ensure this "semantic compatibility", it is enough to solve the two points above. In the following we will disregard this issue, i.e. we will implicitly assume that, for the examples we consider, the institution underlying the semantics of ASL and the "logical framework" of TOOL are "naturally" compatible (at least for some class of specifications).
4. Even if we know how to solve the "semantic compatibility" issues, we still have to face the syntactic aspects: in general, ASL and the TOOL will have different choices w.r.t. the concrete syntax for signatures and formulas, they will not have the same rules w.r.t. the use of overloaded operation names, of infix and mixfix syntax for operation names, etc.

   Clearly, here what we need is some kind of translation mechanism from one concrete syntax to another. Better, we need to define some kind of "specification interchange format" that could be conveniently used as an exchange means. According to the above discussion of the "flattening" problem, we claim that this "specification interchange format" should include (some kind of) "specification-building" mechanisms, in order to be able to reflect the original structure of SP and to let us handle the "flattening" problem on a tool per tool basis. Moreover, some tools use the structure of the specification, and this structure should therefore not be thrown away.

The aim of the next sections is to describe the "specification interchange format" we have designed in the framework of the SALSA project and to further discuss other practical issues raised by inter-operability of tools, on the basis of our

[8] Cf. e.g. the design of the rewrite engine of the ASSPEGIQUE$^+$ specification environment which supports the full PLUSS specification language [25].

experience gained in designing an interface between ASSPEGIQUE$^+$ and the Larch Prover LP [13].

3 Exchanging Specifications

Exchanging specifications between various environments involves two issues: on the one hand, as explained above, some "specification interchange format" has to be used as an exchange means; on the other hand, user interaction related issues must be considered as well, as shown by our former experience when interfacing ASSPEGIQUE and REVE. Let us postpone the user interaction related issues to the next section.

The specification interchange format should be flexible enough to cope with various algebraic specification languages that may considerably differ, for instance in the choice of specification-building primitives (and of their semantics), in the way some pieces of information should be provided explicitly or not by the specifier (e.g. fitting morphisms used in instantiations), in the way two specification expressions are considered to be equivalent (basically, this equivalence of specification expressions can be either "structural" or "nominal", cf. Section 3.2), etc. Hence, the main decision for the design of the specification interchange format is the choice of which entities should be represented.[9] Moreover, special care should be taken in representing explicitly in the specification interchange format all pieces of information "contained" (explicitly or not) in the specification. Finally, pieces of information represented in the specification interchange format should be unambiguous (in particular, terms containing overloaded operators and/or coercions etc.).

Our claim is that the specification interchange format should only be considered as a convenient syntactical means for exchanging specifications. In particular, no semantics has to be associated with this interchange format: given the representation of some specification SP, and knowing in which ASL specification language SP is written, a translator from the interchange format to a given TOOL will be able to reflect in an appropriate way the semantics of SP, provided that all pieces of information that could be relevant in some translation are carefully reflected in the interchange format. Hence the responsibility for ensuring the correctness of the translations performed is left to the translators. This viewpoint is consistent with the discussion in the previous section where we suggested to handle the "flattening" problem on a tool per tool basis.

[9] Practical issues such as the choice of a specific concrete syntax for the specification interchange format and the choice of technologies and tools to implement the translations from and to the specification interchange format are obviously of secondary interest. In what follows, we use examples written in the concrete syntax of the specification interchange format designed in the framework of the SALSA project, but it should be clear that alternative techniques could be considered (e.g. using SGML attributes as in the Z standard interchange format instead of our indexes).

3.1 Specification Interchange Format

The specification interchange format described below has been designed starting from two existing languages and their associated environments (Asspegique$^+$/ Pluss and Lpg [3, 2]) with an additional language/environment in mind (Glider [14]). It is worth mentioning that Glider was still evolving while the interchange format was designed so not all its features were known. Hence Glider provided us with a way to test whether the interchange format designed was flexible enough to handle a new language. Note that Pluss and Lpg strongly differ in various respects: Pluss specifications have a stratified loose semantics while Lpg specifications have an initial (free constructor) semantics, in Pluss there are no implicit instantiations of parameterized specifications, in Lpg specification morphisms are "first-class citizens", there is no renaming primitive in Lpg, etc.

The purpose of the interchange format is to force to describe the structure and content of a specification module – or rather of the set of modules that compose a structured specification – in an explicit way, by splitting the information into elementary pieces which can be later on more easily retrieved by and adapted to the tools that will be applied to the specification. Another constraint is to have a clean separation between the concepts (sorts, operations, signature morphisms, etc.) shared by all languages and the lexical conventions (e.g. naming of sorts, syntax of operations) and contextual restrictions (e.g. kind of overloading allowed, scope rules for operations and sorts) that vary greatly with the environments. The latter differences can then be more easily overcome in a more or less transparent way for the user, as explained in the next section. Hence, we have to define an intermediate representation of specification modules, but in a more detailed way than the one which is usually necessary in the representation designed for – and fitted to – a specific language. In addition to the core of the representation shared by all languages, we will see that it is still possible to add some part in which will be expressed more specific features that are too peculiar to be put in the common part of the representation. Restricting the interchange format to the part that makes sense to all languages and tools would be unnecessarily limiting and it is therefore preferable to have a way to express additional features: it is up to the translator from the interchange format to a given tool to select which parts of this information can be supported, the others being ignored while translating the module.

The description of a module is composed of two main parts, plus an initial "header" (identification of the specification language, author, date of creation, etc.). The first main part lists the components introduced by the module: sorts, operations, axiomatization.[10] The second main part deals with imported modules: it expresses how a given module is related to other modules via the

[10] Let us stress that this part will contain both the components introduced explicitly by the module, i.e. the components that figure textually in the module, and the components introduced implicitly, e.g. as the result of some signature morphism.

specification-building primitives used in the module. A key point to note here is that although the semantics of such specification-building primitives vary greatly with the specification languages, they all rely on a notion easy to describe: the notion of signature morphism. It is not the role of the interchange format to express the semantics, we merely have to give the possibility to retrieve this semantics by describing the signature morphisms and the impact of a specification-building primitive on the components (e.g. its effect on the visibility of sorts or operations). The precise semantics of the specification-building primitives – defined in the source specification language ASL – will be taken into account by the translators. To ease the description of the interchange format we shall first consider only the description of sorts, operations, etc, and illustrate it with the example of a module built on top of another module via an enrichment. Then, we shall describe the part related to the specification-building primitives and illustrate it with more elaborate examples.

The local signature and axiomatization: These parts, introduced by the keyword `%LOCAL`, give the increments in signature and axiomatization provided by an individual module. The new sorts, operations, etc., introduced by a module are listed with an appropriate format. To bypass the problem of knowing how to "name" a component of a module – for instance what is the name of an overloaded operation with a mixfix arity ? – an index is associated with each component: there are indexes for modules, sorts, operations, terms, etc. To describe a component, we select an index for it and list its characteristics in regard of this index. In latter parts of the module, only the index is mentioned. The characteristics for a sort is its name, for an operation it is its syntax and its arity. Note that to fully describe a component – for instance the arity of an operation – we sometimes have to refer to the components of other modules. The sorts and operations from imported modules that are needed to describe the local components – and only those – are listed in the `%EXTERNAL` parts, and the index of their declaring modules must be provided so that we know where to find additional information about these components. Similarly, we describe terms using the indexes of the operations involved and then axioms (and "theorems" supposed to follow from the axioms) using the indexes of terms.

In addition to the index and basic characteristics of a component some attributes can – and sometimes must – be associated with it. For an operation we must state whether it is a generator or a "defined operation" or a variable. For an equation we can mention whether it can be oriented into a rewrite rule (`RR`). The list of attributes is not fixed, although there is an initial list. All environments agree for instance on the notion of generator, while some of them might want to split "defined operations" into "predicates" and non boolean-valued operations. Similarly it is easier to leave as an attribute the "kind" of a module (in Pluss "basic spec" – `bspec` – modules have an initial semantics while modules of kind `spec` do not; in Lpg the kinds of modules are entirely different) than to try to invent an exhaustive list or an unintuitive mapping. The adequate handling of

```
%INFO
  %FISC 1.0
  %LANGUAGE PLUSS
    %RELEASE STANDARD
  %AUTHOR fv
  %DATE 93-11-10-10:40:01
%MODULES
  %D0 %NAME CONTENT spec
  %REFERRED
    %D2 %NAME NAT spec
    %D1 %NAME BOOL bspec
  %CONTEXT %D2 %D1
%SORTS
  %LOCAL    %S1  "Content"
  %EXTERNAL %S2  "nat"   %D2
            %S0  "bool"  %D1
%OPERATIONS
  %LOCAL    %O0  "Cn _"    %S2 %S1 GEN
            %O1  "_ eg _"  %S1 %S1 %S0 PRED
            %O2  "n1"      %S2 VAR
            %O3  "n2"      %S2 VAR
  %EXTERNAL %O4  "_ eg _"  %S2 %S2 %S0 %D2
%MORPHISMS
%VISIBILITY %HIDDEN %O2 %O3
%TERMS
  %T1  %( %O1(%O0(%O2) %O0(%O3)) %)
  %T2  %( %O4(%O2 %O3) %)
%AXIOMS
  %A0 %NAME ax0001 RR %PLUSS %EQUATION %T1 %T2
%THEOREMS
%END
```

spec: CONTENT;
use: NAT;
sort: Content;
generated by:
Cn _ : nat $\rightarrow$ Content;
predicate:
_ eg _ : Content Content;
axioms:
(Cn n1) eg (Cn n2) = n1 eg n2;
where: n1, n2: nat;
end CONTENT;

Fig. 1. A PLUSS module and its associated representation.

a module depending on its kind is better left to the translators.

The description of sorts, operations and axioms is illustrated in Fig. 1, in which the part related to modules can be ignored. Note the subpart for the scope rules (`%VISIBILITY`): in the PLUSS language, variables are not visible outside of the modules where they are declared and are therefore listed as being `%HIDDEN`, an indication for the tools that they are no longer part of the signature accessible to the user. By default, components are visible. Other features of the example displayed in Fig. 1 are explained below.

```
property Function
sorts dom, codom
operators
 f : dom -> codom
satisfies
 Formal_Sort[dom],
 Formal_Sort[codom]
end Function
```

```
enrichment Alpha
 requires
  Function[t1,t2 operators f]
operators
 alpha : seq[t1] -> seq[t2]
variables
 x:t1
 s:seq[t1]
equations
 1: alpha(nil:seq[t1])
      ==> nil:seq[t2]
 2: alpha(x <+ s)
      ==> f(x) <+ alpha(s)
end Alpha
```

```
%MODULES
  %D0 %NAME Alpha %GENERIC enrichment
  %PAR %D2 %D1 %M0
  %REFERRED
    %D1 %NAME Function property
    %D3 %NAME Seq %GENERIC type
    %D4 %D3 %M2
    %D5 %D3 %M4
    %D6 %NAME Elem property
  %CONTEXT %D2 %D4 %D5
%SORTS
  %DERIVED  %S0 "t1"        %D2
            %S1 "t2"        %D2
            %S2 "seq[t1]"   %D4
            %S3 "seq[t2]"   %D5
  %EXTERNAL %S4 "dom"       %D1
            %S5 "codom"     %D1
            %S6 "seq[elem]" %D3
            %S7 "elem"      %D6
%OPERATIONS
  %LOCAL    %O1 "x"        %S0 VAR
            %O2 "s"        %S2 VAR
            %O0 "alpha(_)" %S2 %S3 OP
  %DERIVED  %O3 "f(_)"     %S0 %S1 %D2
            %O4 "nil"      %S2 %D4
            %O5 "_ <+ _"   %S0 %S2 %S2 %D4
            %O6 "nil"      %S3 %D5
            %O7 "_ <+ _"   %S1 %S3 %S3 %D5
  %EXTERNAL %O8 "f(_)"     %S4 %S5 %D1
            %O9 "nil"      %S6 %D3
            %O10 "_ <+ _"  %S7 %S6 %S6 %D3
%MORPHISMS
  %M0 %RENAMING %D1 %D2
       %S4 -> %S0
       %S5 -> %S1
       %O8 -> %O3
  %M1 %FITTING %D6 %D2        |      %M3 %FITTING  %D6 %D2
       %S7 -> %S0             |             %S7 -> %S1
  %M2 %INST %M1 %D3 %D4       |      %M4 %INST %M3 %D3 %D5
       %S6 -> %S2             |             %S6 -> %S3
       %O9 -> %O4             |             %O9 -> %O6
       %O10 -> %O5            |             %O10 -> %O7
```

Fig. 2. Modules `Alpha` and `Function` in LPG and (part of) the representation for `Alpha`.

```
spec: NAME-SET;
as: SET-OF(ITEM ⇒ NAME by m);
where: m: Item ⇒ Name;
renaming:
  Set into Name-set,
  _ belongs to _ into _ is in _ ;
end NAME-SET
```

```
%MODULES %D0 %NAME NAME-SET spec
  %REFERRED %D1 %NAME SET-OF %GENERIC spec
            %D2 %NAME ITEM par
            %D3 %NAME NAME spec
            %D4 %NAME BOOL bspec
            %D5 %D1 %M1
            %D6 %D5 %M2
  %CONTEXT %D3 %D4 %D6
%SORTS %DERIVED  %S0 "Name-set" %D6
                 %S5 "Set"      %D5
       %EXTERNAL %S1 "Name"     %D3
                 %S2 "Bool"     %D4
                 %S3 "Set"      %D1
                 %S4 "Item"     %D2
%OPERATIONS %EXTERNAL %O3  "_ belongs to _" %S4 %S3 %S2 %D1
                      %O4  "the empty set"  %S3 %D1
                      %O5  "_ plus _"       %S3 %S4 %S3 %D1
            %DERIVED  %O0  "_ belongs to _" %S1 %S5 %S2 %D5
                      %O1   "the empty set" %S5 %D5
                      %O2  "_ plus _"       %S5 %S1 %S5 %D5
                      %O6  "_ is in _"      %S1 %S0 %S2 %D6
                      %O7  "the empty set"  %S0 %D6
                      %O8  "_ plus _"       %S0 %S1 %S0 %D6
%MORPHISMS %M0 %NAME m %FITTING %D2 %D3
               %S4 -> %S1
           %M1 %INST %M0 %D1 %D5
               %S3 -> %S5
               %O3 -> %O0
               %O4 -> %O1
               %O5 -> %O2
           %M2 %RENAMING %D5 %D6
               %S5 -> %S0
               %O0 -> %O6
               %O1 -> %O7
               %O2 -> %O8
```

Fig. 3. Module `NAME-SET` in Pluss and (part of) its representation.

The specification-building primitives: In Fig. 1 the part related to the specification-building primitives is quite simple. Since our example is a simple enrichment no signature morphism is involved (so the `%MORPHISMS` part is empty).[11] In other examples this part can be richer. Let us illustrate it with two examples: the first one (cf. Fig. 2), in LPG, is a formal instantiation with an implicit renaming of the parameter (modules of kind `property` are generic), the second example (cf. Fig. 3) is an instantiation in PLUSS combined with a renaming. In both cases we have to express the fitting morphism as well as the corresponding instantiation morphism.

The `%MODULES` and `%MORPHISMS` parts must give an accurate description of how the current module is built on top of other modules. A basic assumption is that the modules are always relative to a notion of a library, in which the combination of the name and kind of a module is always sufficient to refer to it unambiguously. Then, simply referring to a module in the `%REFERRED` subsection corresponds to a simple enrichment. Note that all imported modules must be listed explicitly (even if they are not mentioned in the text of the specification): this is the case of the predefined `BOOL` module in PLUSS. A special subsection (headed by `%PAR`) lists the formal parameters in the case of a generic specification. Sometimes, what a module imports is not a named module from the library but the result of a construction (i.e. a specification expression), for instance a module that is a renaming "on the fly" of another module. In this case, the imported module is described as being the image of the source module by a morphism; in Fig. 2 this is the case of the parameter (to be read `%D2 = %MO(%D1)`) since the sorts `dom` and `codom` of `Function` are renamed in the parameter of `Alpha`. The source module can be either a module from the library or the result of a previous construction. Therefore we can describe complex constructions using auxiliary modules if needed. Only one morphism is given at each step.

A special subsection, headed by `%CONTEXT`, describes which of the modules are visible once the whole construction is complete. This is crucial since there is otherwise no way of knowing which modules are auxiliary and which modules are "real" ones. An example easy to understand is the one of an instantiation of a generic specification in PLUSS: once the instantiation is performed the generic specification and the associated formal components must no longer be accessible to the user. The generic module is therefore not listed in the `%CONTEXT` part. The same is true for a renaming morphism: only the module with the renamed objects is accessible to the user. In Fig. 3, only `%D6` is listed, but neither `%D1`, the generic module, nor `%D5`, the result of the instantiation but prior to the renaming, do appear on that list. Another language might perfectly opt for a different choice and this would still be expressible in the interchange format.

[11] More precisely, the signature morphism is just a signature inclusion and can therefore be left implicit. Leaving implicit such inclusion signature morphisms is the only exception to our "everything is explicit" rule.

The core of the signature morphisms is given in the `%MORPHISMS` section, where the source and target modules are recalled as well as the kind of the morphism involved (and the corresponding fitting morphism for an instantiation morphism).

In Fig. 2 note that the implicit use of the module for sequences has been made explicit (module `%D3`), and similarly the sorts `elem` and `sort[elem]` and their associated operations are given in the `%EXTERNAL` section. The `%DERIVED` parts list the image of existing components through one of the signature morphisms. Finally, in the `%PAR` section, note that the formal parameter is in fact the image of the library module `Function` by a renaming that maps `dom` to `t1` and `codom` to `t2`. The corresponding construction would not be valid in PLUSS where a formal parameter has to be a library module.

Extending the format to take into account specific features: Although all algebraic specification languages deal with the same kind of concepts (sorts, operations, morphisms, etc.) it is not unusual for a particular language to attach a more "operational" content to some parts, or to be richer for some other parts. For instance, in GLIDER it is possible to associate a set of axioms to "the" operation they define while in PLUSS and LPG there is no such correspondence. In LPG additional morphisms between modules can be recorded to ease subsequent instantiations. Hence, it is useful to let each language state these additional features, even if not all tools can make use of them. The extension of the kernel of the interchange format shared by all languages and tools is achieved in two ways. A first mechanism is the possibility to associate attributes with components like sorts, operations, etc. The attributes have a lexical form different from the one of the other parts of the interchange format, so it is quite easy to handle them during the parsing of a specification translated into this interchange format and to filter the attributes that have no meaning for a given target tool. The second way is by having subparts (in fact sub-grammars) in the syntax of the interchange format: one for each specific specification language. It has proved quite easy to define a part specific to some language using the basic components (index, attributes) already existing, perhaps with some additional keywords, and to adapt the translators when a specific part must be extended or a new one created when taking into account some additional specification language not considered so far.

3.2 Producing and Reading the Specification Interchange Format

Translating a specification from a given language into the interchange format is the easiest of the two translations involved in the exchange of specifications. It amounts to listing the components in the right format and order, and to explicitly storing the morphism constructions associated with the specification-building primitives. With respect to existing languages and environments, the effort for producing the interchange format is basically limited to additional traversal of the internal representations of modules and some extra book-keeping with respect to what is usually needed. Translators from Glider, Lpg and Pluss into the specification interchange format have been implemented.

Reading a specification translated in the interchange format is more challenging since this is where both syntactic transformations and some kind of reshaping of the specification must be performed. An example of a simple syntactic transformation corresponds to the use of different lexical conventions between the source specification language ASL and the target Tool. A mapping between old and new names must be provided, at least in a first approach, although this may not be transparent for the user (cf. Section 4). Besides these straightforward problems and similar ones – like providing customized equivalents to the predefined items of the source specification language ASL, if any – there are other problems related to differences in semantics and also in the collection of specification-building primitives that may vary from one language to another.

An important point to note – that makes the translation more realistic than it can appear at first sight – is that we are not looking for a source to source translation but we need only to mimic for a given Tool the effect of the primitives of the source specification language ASL. When no accurate source to source translation would exist, sometimes it is possible to simulate an equivalent effect directly at the level of the Tool. As an example, consider the problem of simulating the renaming primitive that exist in Pluss but not in Lpg. No translation into a valid Lpg text can exist, while acting directly on the signature available during a Lpg session might be feasible! The same is true when inputing a Pluss specification into something readable by the Larch Prover's parser for which there are no true parameterization/instantiation primitives. The translation can sometimes involve some transformation of the structure of the specification: adding auxiliary modules, or factoring some parts. An example of the first transformation occurs with the Lpg module `Alpha`. Implicit renaming – like the one of `Function` – is not available in Pluss so an explicit module for the renaming must be created while reading the representation of `Alpha` into the interchange format. Reshaping of a specification is needed because Lpg and Pluss differ in the way they handle the equivalence of specification expressions. Borrowing programming language terminology, the equivalence of specification expressions in Lpg can be coined as "structural" since multiple instantiations are viewed as being the same as long as the underlying morphisms coincide. On the contrary the equivalence in Pluss can be coined as "nominal" since the instantiations must be associated with an explicit module and named, and only names of modules matter when testing if two specification expressions are the same. Therefore, reading a valid Lpg specification containing multiple "equivalent" instantiations of a generic specification would result in an erroneous result for a tool from the Asspegique$^+$ environment since a given component would seem to be defined in more than one module, something which is considered as a structuring error in Pluss: when reading the Lpg specification translated in the interchange format we have to reshape (when possible) the specification to factor and name that instantiation. This situation would occur if, for instance, in some LPG module `Mod` we further instantiate `Alpha` so as to map the sorts `t1` and `t2` to the same sort, say `nat`. The representation of the LPG module `Mod` would contain two equivalent instantiations of `seq` resulting in two equivalent occurrences of `seq[nat]`. In PLUSS the instantiation has to be made once and for all in a separate module that may be imported twice afterwards in the instantiation of `Alpha`. The role of the interchange format is to ease the detection of such situations by describing the constructions in a very detailed and elementary way. Handling the situation – and deciding where to put the limit in this reshaping of the specifications – is left to the translators.

Currently, we have implemented two translators from the interchange format. The first one reads modules translated in the interchange format and translates them directly into the internal representation of PLUSS modules in the ASSPEGIQUE$^+$ environment. For instance, the specification rooted by the LPG module `Alpha` has been input via the interchange format into the rewriting engine available in the ASSPEGIQUE$^+$ environment, modulo some translation of names (e.g. `nil` is a forbidden identifier, `seq[t1]` is not a valid sort notation, etc.). The second translator is a first draft of a gateway between ASSPEGIQUE$^+$ and LP, the proof assistant of the Larch project. From the interchange format, a text file is generated that can be input in LP's parser. However, for this gateway to be really usable, additional user interaction problems must be solved, since user interaction issues are much more sophisticated for an interactive proof assistant than for an automatic tool like a plain rewriting engine. These user interaction related issues are the topics of the next section.

4 Interaction with the User: Getting and Displaying Information

Any tool usually requires an interaction with the user, both for the purpose of reading some input issued by the user, and for the purpose of displaying results. For instance, a theorem prover reads the theorem the user wants to prove, and displays informations about the proof steps, or asks for some proof command. The content of the messages read or displayed involves both the specification signature and term language, the formula language, and the tool command language; therefore parsing/unparsing these messages involves these different languages. For example:

```
prove x=y ⇒ prefix(cons(x,cons(2,empty))) = prefix(cons(y,empty))
```

is an input to the Larch Prover LP, where "`prove`" belongs to the command language, "⇒" and "`=`" to the formula language, and "`x`", "`y`", "`prefix(cons(x, cons(2,empty)))` and "`prefix(cons(y,empty))`" to the term language. Similarly:

```
Current conjecture user.1:
  x = y ⇒ prefix(cons(x,cons(2,empty))) = prefix(cons(y,empty))
```

`Proof suspended.`
is an example of output by LP, that includes some text, e.g. "`Current conjecture user.1:`", in addition to a formula.

Here we assume again that the user is working with an algebraic specification SP written in a specification language ASL, and wants to use some TOOL. Moreover we assume that all the information about the "content" of SP has been converted into the specification interchange format and successfully input to the TOOL (possibly after some adequate translation, cf. Section 3.2). The next issue to be addressed now is how the user will be able to interact with the TOOL, without having to take care of the necessary translations. In particular, our aim is to design simple mechanisms that will make the translations as transparent as possible for the user (in contrast with our first attempt with the ASSPEGIQUE/REVE interface, cf. Section 1).

4.1 Handling TOOL's Input

Let us first consider the issues related to the input issued by the user. Since the command language is specific to the TOOL used, in its input the user will necessarily have to use this command language as defined by the TOOL (this is not a problem, since to use the TOOL the user must have a minimal knowledge of the TOOL anyway). On the other hand, if we want the required translations to be transparent for the user, then all occurrences of terms, formulas, etc. in the input should be expressed using the original syntax as defined by ASL and SP (hereafter the corresponding language is called the ASL/SP language). This means that the user input is then a combination of the TOOL command language and of the ASL/SP language, and unfortunately this combination cannot be parsed by either the parser designed for ASL or the parser of the TOOL. Even if it were, the result would be meaningless, since the signature the TOOL knows is the signature resulting from some translation of the specification SP converted to the specification interchange format. Therefore, to achieve our goal, we have to proceed as follows:

1. The user input is written in an appropriate combination of the TOOL command language and the ASL/SP language. There should exist some easy way to "split" this input into homogeneous pieces of either the TOOL command language or the ASL/SP language.[12]
   For instance, using the ASSPEGIQUE$^+$/LP interface, the example given at the beginning of this section would be written by the user as:
   `prove "x = y ⇒ prefix of x 2 empty = prefix of y empty"`
   (assuming that a mixfix operation "`prefix of`" is declared in SP, as well as an "anonymous cons" operation).

[12] Depending on how the TOOL language and parser are designed, this splitting task can be easy or may require that the user input is marked with some additional delimiters. In the ASSPEGIQUE$^+$/LP interface, the pieces expressed in the PLUSS/SP language are enclosed in quotes.

2. Pieces expressed in the ASL/SP language are parsed using the parsing tools designed for ASL. The result is then converted into the specification interchange format, using of course the adequate indexes, and then again translated into pieces suitable for the TOOL. For this, we just have to assume that we have kept track of the correspondences SP-name ↔ index ↔ TOOL-name. Then the pieces corresponding to the TOOL command language and the pieces corresponding to the converted+translated parts are put together again, and can be used as a suitable input for the TOOL.
   For instance, in our example, the intermediate result expressed with indexes could be something like:
   ```
   %IMPLIES(%EQ(%010 %011)
             %EQ(%05(%02(%010 %04(%07 %01))) %05(%02(%011 %01))))
   ```
   and the final result in LP's syntax would be:
   ```
   prove 010 = 011 => 05(02(010,04(07,01))) = 05(02(011,01))
   ```
   (here the final translation from indexes to LP-names is very simple, we just have to omit the "%", which is used for comments in LP, and adopt a standard prefix notation).[13]

Note that since each specification language has its own conventions with respect to the use of overloaded operations, coercions, anonymous operations, mixfix syntax etc., it seems impossible to design a general purpose parser that will faithfully reflect the conventions of the chosen ASL specification language. In particular, it is often the case that the term language is ambiguous, and each specification language has its specific rules for handling the possible ambiguities. This is the reason why we decide to use the parser designed for the ASL specification language, and to implement the cooperation scheme outlined above between this parser and the TOOL's parser.

4.2 Handling TOOL's Output

Once the issues related to the user input are solved, it is now quite easy to understand how we will proceed with respect to the output of the TOOL. Indeed we just have to assume that this output will be split again into homogeneous pieces (pieces of text specific to the TOOL on one hand, terms/formulas/operation names on the other). A symmetric translation to the one described above for the input is then performed on the term/formula pieces (e.g. a formula expressed in the TOOL syntax is first converted to a formula expressed in the specification interchange format using indexes, and then further translated to a formula expressed in the ASL/SP formula language).

There are still some difficulties to overcome when the TOOL integrates some clever pretty-printing mechanism (as it is the case for LP). Let us call "raw output" the output produced by the TOOL before any reverse conversion+translation

[13] Since the actual operation names used by LP will be hidden to the user, there is no need to try to reflect the operation names as they are given in SP, and it is much easier to choose a very simple translation scheme.

takes place. Pretty-printing this "raw output" is not convenient, since once the reverse conversion+translation will have been performed on the term/formula pieces, the length of the pieces will in general have considerably changed (e.g. in our example, "`O5`" is converted into "`%O5`" and then translated into "`prefix of`", etc.). Again, the solution we suggest is to implement some cooperation scheme between the TOOL's pretty-printer and the pretty-printer designed for ASL:

1. TOOL will not perform the pretty-printing of its output, **but** will use its pretty-printing facilities (and its knowledge of the logical structure of the output) to produce a "marked-up output", i.e. the raw output to which formatting indications are added (thereby the term/formulas pieces are considered as single units, and will not be enriched by any formatting indications).
2. The reverse conversion+translation is performed, and appropriate formatting indications for the term/formula parts are added to them, by the pretty-printing facilities of ASL.
3. Then all pieces are put together, the result being processed by the display algorithm of the TOOL's pretty-printer.

Note that this solution requires that both pretty-printing facilities use the same convention for formatting indications.

4.3 Signature Evolution

Up to now, we have assumed that the TOOL uses the information (signature, axioms, etc.) relative to some specification SP in a *static way*. However, some tools, such as e.g. theorem provers, may need for instance to declare some new variables or constants in the process of a proof, which implies that the signature used by the TOOL is *dynamically* modified. In such cases it is necessary to reflect this modification on the signature used by the ASL's parser to parse terms and formulas (as explained in Section 4.1), since the user may have to refer to the newly created variables or constants in further proof commands. In order to do that, what is needed is:

1. A protocol between the TOOL and the ASL supporting environment for exchanging the information related to signature updates.
2. To keep track in the correspondences SP-name $\leftrightarrow$ index $\leftrightarrow$ TOOL-name of the signature updates.

Implementing these mechanisms is not very difficult. However, when looking more closely to the details, we may need a more elaborate protocol to reflect, at the user level, the specific TOOL's strategy for choosing "new names" (if at all possible). For instance, when LP creates a new constant from a variable "`z`" (e.g. when performing a proof by induction on `z`), it tries to call it "`zc`", to help the user remember the origin of this new constant. However, in our case, due to our conversion+translation scheme, the variable denoted "`x`" by the user is denoted "`O1O`" by LP.[14] Hence we should manage that the new constant introduced by LP, and named "`O1Oc`" by LP, is reflected to the user as a new constant named "`xc`", provided "`xc`" is indeed a "fresh name" (as was "`O1Oc`" on LP's side). We have implemented such a protocol for the ASSPEGIQUE$^+$/LP interface (i.e. our protocol includes messages such as "new const `O1Oc` derived from var `O1O`"), but it is clear that in some cases it will not be possible to reflect in a faithful way the TOOL's strategy for choosing new names (e.g. in the case of a specification language where all names starting by "x" denote variables...).

More generally, it may be necessary to enrich the protocol between the TOOL and the ASL supporting environment to keep track of the results generated by the TOOL. For instance, it would be interesting to "enrich" the specification SP with e.g. the theorems proved with the TOOL, etc. Clearly, more work is required to investigate these issues.

As a last remark, we would like to point out that with the approach described in this section, the user will be able to operate TOOL in a fully transparent way, and can ignore all the translations that take place behind the scene. Our approach requires, however, some modification of the TOOL's code (mainly to add the entry points to the specific code implementing the cooperation schemes), but we claim that these modifications are both rather minor and in principle well localized (e.g. no modification of the proof engine is necessary). This approach has been implemented and successfully experimented with the ASSPEGIQUE$^+$/LP interface.

5 Conclusion

We have presented in this paper our technical approach to enable the use of various tools available for algebraic specifications without requiring the user to translate his(her) specifications in the different languages supported by these tools. Let us recall that our goal is not to build new tools, but to be able to **reuse** existing ones, while keeping the implied coding task minimal.

We first described the required compatibility between the logical framework of the tool and the institution underlying the semantics of the specification language used to express the user specification. We then presented the main issues to be considered for the design of a generic technical solution to this interoperability problem. A first issue is the communication of data in an adequate format: for this purpose, a specification interchange format was designed, and translators to and from this format were implemented for several specifications languages (PLUSS, LPG, and GLIDER). User interaction then requires special care to ensure that the underlying translations are transparent. When using for instance theorem proving tools (e.g. LP), other issues are raised such as dynamic

[14] We assume here again the correspondence `x` $\leftrightarrow$ `%O1O` $\leftrightarrow$ `O1O`.

changes in the signature (for instance, when new constants or new variables are introduced).

The work presented here results from our experience in building interfaces between our ASSPEGIQUE$^+$ environment and other existing tools, and from the aim to provide a generic design for such interfaces. This work should not be considered as a fixed and final result, but rather as a starting point for a better inter-operability of existing (and future) tools and we hope that it could be used as the basis of further developments in the scope of the Common Framework Initiative [1].

Acknowledgements: Special thanks are due to the anonymous referees of the submitted version of this paper for careful reading and helpful suggestions.
The ideas presented in this paper benefited from numerous fruitful discussions and deep cooperation with our colleagues of the SALSA project (D. Bert, R. Echahed, J.-M. Hufflen, J.-P. Jacquot, M. Lemoine, N. Levy, J.-C. Reynaud, C. Roques), and with our colleagues of the Larch project (S. Garland, J. Guttag, M. Vandervoorde). In particular the specification interchange format described in this paper was elaborated in the framework of the SALSA project, and the design of the ASSPEGIQUE$^+$/LP interface has been done in cooperation with S. Garland, J. Guttag and M. Vandervoorde, with the support of a CNRS/NSF grant. Special thanks are due to M. Vandervoorde who kindly implemented the modifications of LP that were necessary for the ASSPEGIQUE$^+$/LP interface.

References

1. The Common Framework Initiative. Web home page available at the address http://www.brics.dk/Projects/CoFI/, 1995.
2. D. Bert, P. Drabik, and R. Echahed. Manuel de référence de LPG. Technical Report 17, IMAG-LIFIA, 1987.
3. D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *ESOP-86*, pages 119–132. Springer-Verlag L.N.C.S. 213, 1986.
4. M. Bidoit. PLUSS, un langage pour le développement de spécifications algébriques modulaires. Thèse d'Etat, Université Paris-Sud, 1989.
5. M. Bidoit, C. Choppy, and F. Voisin. The ASSPEGIQUE specification environment: Motivations and Design. In *Proc. of the 3rd Workshop on Theory and Applications of Abstract Data Types*, pages 54–72. Springer-Verlag I.F.B. 116, 1984.
6. M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? An experiment with the PLUSS specification language. *Science of Computer Programming*, 12(1), 1989.
7. M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development — A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
8. R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. In *Advance Course on Abstract Software Specifications*, pages 292–332. Springer-Verlag L.N.C.S. 86, 1980.

9. M. Cerioli. Relationships between Logical Formalisms. PhD Thesis, TD-4-93, Universita di Pisa-Genova-Udine, 1993.
10. C. Choppy. ASSPEGIQUE: User's Manual. L.R.I. Research Report 452, 1988.
11. C. Choppy. About the "correctness" and "adequacy" of PLUSS specifications. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specifications - 9th Workshop on Abstract Data Type joint with 4th COMPASS Workshop - Selected Papers*, pages 128–143. L.N.C.S. 785 Springer Verlag, 1994.
12. C. Choppy. Spécifications algébriques: Prototypage et validation. Habilitation à diriger des recherches - Université Paris-Sud, 1994.
13. C. Choppy and M. Bidoit. Integrating ASSPEGIQUE and LP. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch*, pages 69–85. Springer-Verlag Workshops in Computing, 1993.
14. S. Clérici, R. Jiménez, and F. Orejas. Semantic Constructions in the Specification Language GLIDER. Technical Report Dissem-033-P, Icarus, June 1992.
15. H. Ehrig, W. Fey, and H. Hansen. ACT ONE: an algebraic specification language with two levels of semantics. Technical Report 83–03, TU Berlin FB 20, 1983.
16. R. Forgaard and J. Guttag. REVE: a term rewriting system generator with failure-resistant Knuth-Bendix, 1984. Proc. of an NSF workshop on the rewrite rule laboratory, and Report n° 84GEN008, General Electric.
17. S. Garland and J. Guttag. An overview of LP, the Larch Prover. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151. Springer-Verlag L.N.C.S. 355, 1989.
18. S. Garland and J. Guttag. A Guide to LP, The Larch Prover. Technical Report 82, DEC-SRC, 1991.
19. M.-C. Gaudel. Structuring and modularizing algebraic specifications : the PLUSS specification language, evolutions and perspectives. In *Proc. of the 9th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 3–23. Springer-Verlag L.N.C.S. 577, 1992.
20. J. A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39:95–146, 1992.
21. SALSA (group composed of D. Bert, M. Bidoit, C. Choppy, R. Echahed, J.-M. Hufflein, J.-P. Jacquot, M. Lemoine, N. Levy, J.-C. Reynaud, C. Roques, and F. Voisin). Opération SALSA: Structure d'AccueiL pour Spécifications Algébriques. Final report, December 1993.
22. S. Kaplan and C. Choppy. Abstract rewriting with concrete operators. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, pages 178–186. Springer-Verlag L.N.C.S. 355, 1989.
23. P. Lescanne. Computer experiments with the REVE term rewriting system generator. In *Proc. 10th Symposium on Principles of Programming Languages*, pages 99–108. ACM, 1983.
24. T. Mossakowski. A hierarchy of institutions separated by properties of parameterized abstract data types. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications - 10th Workshop on Abstract Data Type joint with 5th COMPASS Workshop - Selected Papers*, pages 389–405. L.N.C.S. 906 Springer Verlag, 1995.
25. C. Roques. Modularité dans les spécifications algébriques, Théorie et application. Thèse de Doctorat, Université Paris-Sud, 1994.
26. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

27. F. Voisin and M. Bidoit. Towards Modular Theorem-Provers for Modular Algebraic Specifications: Modular Orientation of Equations into Rewrite Rules. Submitted to the proceedings of the 11th WADT, 1995.
28. M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42(2):124–249, 1986.
29. M. Wirsing. Algebraic specification languages: an overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications - 10th Workshop on Abstract Data Type joint with 5th COMPASS Workshop - Selected Papers*, pages 81–115. L.N.C.S. 906 Springer Verlag, 1995.