Simulating Algebraic Specification Genericity on Languages with Initial Semantics.

Martins Anamaria, Santana Anderson

Simulating Algebraic Specification Genericity on Languages with Initial Semantics [1]

Anamaria Martins Moreira [3]

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte — UFRN
Natal, Brasil

Anderson Santana de Oliveira [2,4]

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte — UFRN
Natal, Brasil

Abstract

This paper discusses the concept of genericity often used in algebraic specification languages and how this concept can be simulated in a meta level in languages with purely initial semantics, as it is the case for ELAN and ASF+SDF. This proposal is being integrated into the FERUS tool, in development for ELAN, and will have the effect of providing in the meta level better modularity features without any changes to the language itself, as long as all manipulations are done through the operations available in the tool.

Key words: algebraic specifications, genericity, transformation tools, ELAN, ASF+SDF.

1 Introduction

Modularity is a key feature for reuse, and algebraic specification languages usually provide structuring constructs that support the definition of modular specifications. Additionally, it is known that more general components are more likely to be reusable than specific ones. Algebraic specification languages usually include genericity mechanisms that provide great flexibility in

[1] This work is partially supported by a joint research project funded by CNPq and INRIA.
[2] Supported by CAPES.
[3] Email: `anamaria@consiste.dimap.ufrn.br`
[4] Email: `anderson@consiste.dimap.ufrn.br`

the level of generality of a component by its parameterization. Finally, algebraic specifications, with their formal syntax and semantics, can provide great support for software component reuse, as they allow tools to "understand" the semantics of the components they are manipulating. Briefly, algebraic specifications are well suited for reuse as they provide the characteristics that are needed in the construction of specifications from already existing ones, in the construction of the reusable components and in the implementation of tool support.

The FERUS tool [13,16], which was first developed to deal with CASL [8] specifications, has the goal of supporting algebraic specification components development. It provides an environment for specification design and prototyping, that allows to edit, compile and execute specifications (given that the specification is executable). Its main feature, however, is the possibility to derive new components through reuse driven transformation operations. For example, to create an instance of a parameterized (generic) module using the instantiate operation, and conversely, to create a parameterized (generic) module by abstracting some sorts and related declarations, with the generalization operation.

In order to demonstrate that FERUS is suitable to different contexts and/or languages inside the domain of algebraic specifications, we proposed to adapt the tool to the language ELAN [4,22]. The idea was to keep the tool architecture unchanged, thus only language specific features were subject of adaptation. An interesting feature of applying FERUS to the ELAN language is that although ELAN has many characteristics of an algebraic specification language, it was not conceived as such and there are some major differences in semantics and structuring (modularity) constructs. The flexibility of the tool could then be better tested, with promising results [17].

The FERUS tool works on the new version of ELAN called ELAN 4. This new version borrows the syntax of the ASF+SDF specification language [10,22], as well as its semantics if we do not consider some features of ELAN: we are dealing with most of the language, except for rewriting *strategies* [5]. Its underlying institution and model semantics are the same as the for the ASF+SDF language, i.e., $Cond^{=}$ with initial semantics, as classified by Mossakowsky in [18]. Structuring and parameterization in these languages is very restricted, however. The FERUS tool may then be used to improve these facilities without having to extend the language itself. This is particularly tricky and interesting when it comes to defining genericity in a more standard way (purists in the algebraic domain may say that ELAN and ASF+SDF do not have genericity, since they only allow for initial semantics), and this is where we focus our attention in this paper.

This paper is structured as follows: section 2 gives the main concepts related to genericity in algebraic specifications, section 3 presents the language ELAN and discusses its features with focus on modularization and parameterization, in section 4 we show how genericity can be simulated in ELAN by manipulating specifications in a meta level. The paper concludes with issues related to the integration of this simulation process into the FERUS tool.

2 Algebraic Specifications and Genericity

Algebraic specifications [25] are a classical paradigm for specifying functional properties of systems. It is a simple and well founded technique which presents some nice characteristics such as *executability* (of some particular specifications), usually carried out by rewriting [9].

An algebraic specification usually consists of *sort* and *operator* declarations, defining a *signature*; and *axioms* built over this signature (and some set of variables). Axioms describe properties which are expected to be true in all models of the specification. Signature and axioms together are called the *presentation* of the specification. To this presentation corresponds a *semantics*, which is traditionally presented as the class of algebras that are *models* of the presentation or the set of formulas that constitute the underlying *theory*. Recently, starting with Maude [7,11] and the rewriting logic [14], on one hand, and ELAN and the ρ-calculus [6] on the other hand, some different interpretations of algebraic specifications have been proposed; but it is the classical approach that we consider here: algebraic structures as models, with a set of values for each declared sort, a function for each declared operator, and rewriting as the computational executability tool.

An algebraic specification language is characterized by many different factors, which can in general be identified as the institution over which specification components are written and its modularity constructs. Each different language has its particularities in both aspects. In this work we consider some characteristics that are available in most of the existing languages: conditional equational logic, many-sorted total operators, and the built-in equality predicate as the unique predicate. With this basic building block, we can then have initial and loose semantics (only the initial algebra or all algebras that satisfy the specification axioms). We can also have structured specifications that import previously defined ones with some constraints that define how imported specification models are to be used in the definition of the models of the importing specification.

In most algebraic specification languages, the parameter of a generic specification is supposed or required to present a loose semantics, possibly with some extra constraints, accepting a large class of possible models. Then, instantiation is the substitution of this generic parameter by some or one of its "models" (a specification whose models are a sub-class of the models of the parameter, modulo signature restriction and translation, identified by the instantiation signature morphism). This *specialization condition* is needed for an instantiation to be defined. This is the case, e.g., for LPG [3], CASL [8] and Maude [7]. In all of them, the parameter specification has the goal of specifying some set of "minimal" conditions that must be satisfied by a specification (usually, an abstract data type) to be used as an actual parameter for the generic module. These conditions may be purely syntactic, in the case of a parameter with loose semantics and no axioms (only defining a signature), or syntactic and semantic, if there are any other constraints or axioms. In this latter case, each model of each potential actual parameter must satisfy all axioms or constraints specified in the formal parameter to correspond to a valid instantiation. Furthermore, there is usually an extra condition, called the *instantiation push-out condition*, requiring that the instantiation process do not introduce any new sharing of symbols between the body of the new specification and the actual parameter. These are then the conditions that we are proposing to simulate via the meta operations of FERUS in ELAN and ASF+SDF.

The definition of instantiation that we will be taking into account in the following is given below in definition 2.1. It is a somewhat abstract definition that has to be slightly adapted to each application language, but which deals with the main common features of what is expected to be instantiation in algebraic specification languages.

Definition 2.1 [Instantiation Operation] Given

(i) a specification sp with generic parameter sp_{gp} and body (including imported specifications) sp_b;

(ii) a candidate actual parameter sp_{ap}, and

(iii) a signature morphism m from the signature Σ_{gp} of sp_{gp} into Σ_{ap} of sp_{ap}; then,

the **instantiation of** sp **by** sp_{ap} **with respect to** m is defined if

(i) all models of sp_{ap}, restricted via m into models of the signature Σ_{gp}, are models of sp_{gp}, and

(ii) the instantiation push-out condition is satisfied.

It defines the specification sp' where

- sp_{ap} substitutes sp_{gp}, and is added on top of the list of imports of sp, and
- all local and imported items of sp are renamed according to m.

Note that the specification sp' in the definition above may be explicitly generated, as it happens when we have a meta-operation (e.g., in FERUS), or only implicitly, as it is the case with the corresponding construct in LPG, CASL and Maude. In this latter case, this definition states the application conditions of the instantiation construct and the semantics of the specification that contains it.

This definition roughly corresponds to the classical instantiation diagram (see figure 1) found for instance in [12].

The main variations of the above definition are generally due to differences in the structuring constructs for each language. For instance, in CASL, the

$$\begin{tikzcd}
sp_{gp} \arrow[r] \arrow[d, "m"] & sp_b \arrow[d] \\
sp_{ap} \arrow[r] & sp'_b
\end{tikzcd}$$

Fig. 1. Instantiation diagram

parameter specification may rely on a list of imported specifications, and may not be a stand-alone specification (this is not the case if we only work with named specifications). Also, it is often allowed to provide a mapping of symbols instead of a morphism, as it happens in CASL. This definition takes into account the morphism derived from it, when it can be correctly and uniquely derived. On the other hand, in (full) Maude, every item identification includes its origin, making the push-out condition trivially satisfied. Because this is not the case with ELAN and ASF+SDF, we will need the push-out condition. Also, in (full) Maude, if the actual parameter is still a theory, it is added as formal parameter, instead of import, allowing for further instantiation. All these variations have minor influence on the semantics of the resulting specification, however, and for our purposes, this abstract definition is enough.

3 The ELAN language

ELAN is a specification and prototyping framework for algebraic specifications whose main application areas include theorem proving, constraint solving, and logical programming [4].

It is based on rewrite theory with user defined strategies. Using ELAN's programming language it is possible to define non-deterministic (as well as deterministic) computational systems, as it provides operators for combining conditional rewrite rules; iterators, to control how many times they can be applied; and selection operations corresponding to deterministic and non-deterministic choices of strategies [5].

Maintenance issues related to changes in syntax of ELAN language have demanded more flexibility of the parsing technology under use. This led to the adoption of ASF+SDF syntax, parsing tools, and exchange data format as reported in [23], originating the current version of the environment, called ELAN 4. Consequently, most of the syntax and semantics of ELAN 4 is the same of ASF+SDF with exception of strategies definition and application. For this reason, when we talk here of ELAN (without strategies), one can also think of ASF+SDF and vice-versa.

The new version of the ELAN language presents constructs for definition of lexical and context-free syntax, including sort declarations, character classes represented as regular expressions, production definitions corresponding to the context-free syntax, variable declarations, and finally, rewrite rules definition. Both syntactic and lexical definitions can be exported or hidden, meaning global or local declarations respectively. The following example briefly introduce part of the language our tools deal with. In order to make development cycles faster in an incremental software development paradigm, we have delayed the treatment of specifications containing lexical syntax, user defined priorities, infix operators, and strategies. Of those, only the inclusion of strategies may have a significant impact on the work we present here, because it has significant impact on the semantics of the corresponding ELAN specifications (we do not claim that our work is directly applicable for algebraic specification languages with strategies). The other items cited above, however, may be seen as syntactic sugar and their inclusion in the language does not invalidate any of the results presented here.

Example 3.1 In ELAN, the algebra of natural numbers can be defined using prefix operations as follows:

```
module NATURAL
  imports    BOOLEAN
  exports
    sorts Nat
    context-free syntax
      zero            -> Nat
      succ (Nat)      -> Nat
      add  (Nat, Nat) -> Nat
      eq   (Nat, Nat) -> Bool
    variables
      "x" -> Nat
      "y" -> Nat
  rules
    [] add(zero, x)         => x
    [] add(x,succ(y))       => succ(add(x,y))
    [] eq(zero,zero)        => true
    [] eq(zero,succ(x))     => false
    [] eq(succ(x),zero)     => false
    [] eq(succ(x),succ(y)) => eq(x,y)
```

In the example above BOOLEAN is assumed to be a specification for the boolean algebra, with contents that can be seen in example 3.2. Additionally, in ELAN 4 all modules with rules need to import a predefined specification ElanLayout, that we omit here. However, this specification only contains definitions of what should be regarded as syntactic layout, like whitespace and comments, and does not have any influence in the semantics of the resulting specification.

In the next section we present in some more detail the modularization constructs available in ELAN and ASF+SDF, as it is the point that we are trying to improve with our proposal.

ASF+SDF has as modularization constructs: *imports*, *renamings* and *parameterization* [24]. Renamings contribute to the reuse of specifications, adapting identifiers in a module to their new context. It states that all occurrences of the items to be renamed are to be replaced by the corresponding symbols in the renaming map. The purpose of renamings is to adapt the identifiers of a certain module to a different context, and also to avoid name clashes. On the other hand, parameterization provides the possibility to create reusable modules through the declaration of formal parameters, but in a restricted way, since parameters are seen just as a list of symbols (sort names) that have to be replaced. In fact, parameterization is a special case of renaming.

When considering the semantics of ASF+SDF (borrowed by ELAN 4) one may find in [2,19] that it is defined over the normalized form of a structured module: "the semantics of module m in the context of the specifications S is the initial algebra of its normal form $N(m, S)$, provided the latter has no void sorts and no unbound parameters" [2]. This normalization process consists in removing all syntactic sugar and modularization, generating a "flat" module in an abstract syntax format. For the example 3.1 above, one would get a specification like the one in example 3.2. The main consequence is that there is no hierarchical semantics organization, in the sense that one can not compose the semantics of a module from the initial models of its sub-modules.

Example 3.2 The normal form representation of the example 3.1 is:

```
module NATURAL
  exports
    sorts Nat
          Bool
    context-free syntax
      zero            -> Nat
      succ (Nat)      -> Nat
      add  (Nat, Nat) -> Nat
      eq   (Nat, Nat) -> Bool

      true              -> Bool
      false             -> Bool
      or  (Bool , Bool) -> Bool
      and (Bool, Bool)  -> Bool
      not (Bool)        -> Bool

    variables
      "x" -> Nat
      "y" -> Nat
      "B1" -> Bool
  rules
    [] add(zero,x)          => x
    [] add(x,succ(y))       => succ(add(x,y))
    [] eq(zero,zero)        => true
    [] eq(zero,succ(x))     => false
    [] eq(succ(x),zero)     => false
    [] eq(succ(x),succ(y))  => eq(x,y)

    [] and(true,B1)  => B1
    [] and(false,B1) => false
    [] or(true,B1)   => true
    [] or(false,B1)  => B1
    [] not(true)     => false
    [] not(false)    => true
```

4 Simulation of Genericity in ELAN

Because there is no loose interpretation of specifications in ELAN and ASF+SDF (only initial semantics), genericity cannot be treated in the same way as described in section 2. Parameterization and instantiation in these languages are just special cases of the more general *imports* and/or *renaming* constructs, and there is no specialization or push-out requirements in the instantiation of a "generic" parameter. We propose then to use a "meta-genericity" pattern via meta operations of instantiation and generalization[5] to obtain this semantically more controlled behavior of genericity and instantiation found in some algebraic specification languages.

In the following, we present in more detail, through an example, what can be done in ELAN, the related problems and what we propose in substitution to this.

Example 4.1 Consider the ELAN 4 specification of lists with an accumulation operation below where variable declarations have been omitted:

```
module LIST[Elem]
  exports
    sorts   Elem List[[Elem]]
    context-free syntax
      k                           -> Elem
      bin (Elem, Elem)            -> Elem
      nil                         -> List[[Elem]]
      cons (Elem, List[[Elem]]) -> List[[Elem]]
      sum (List[[Elem]])          -> Elem
```

[5] The generalization operation [15] is used to generate generic specifications from non-generic ones, contributing to their reusability. It is briefly described in section 5.

```
  rules
    [] sum(nil)       => k
    [] sum(cons(x,l)) => bin(x, sum(l))
```

Note that this specification does not have a semantics, as explained in section 3.1 (its parameters must be instantiated first). On the other hand, we would like to specify it in a more modular way, separating the generic part definition from the fixed part that is constructed from it, by something like

```
module LIST[PARAM]
  exports
    sorts List
    context-free syntax
      nil                -> List
      cons (Elem, List) -> List
      sum (List)         -> Elem
  rules
    [] sum(nil)       => k
    [] sum(cons(x,l)) => bin(x, sum(l))
```

with

```
module PARAM
  exports
    sorts Elem
    context-free syntax
      k                  -> Elem
      bin (Elem, Elem)  -> Elem
```

But this is not allowed in ELAN (parameters cannot be modules, but only sorts). We can import `PARAM` in the specification `LIST`, as shown below,

```
module LIST
  imports    PARAM
  exports
    sorts List
    context-free syntax
      nil                -> List
      cons (Elem, List) -> List
      sum (List)         -> Elem
  rules
    [] sum(nil)       => k
    [] sum(cons(x,l)) => bin(x, sum(l))
```

but we would not get the desired semantics, because we would only have isomorphic models to the term algebra, with values `k`, `bin(k,k)`, `bin(k,bin(k,k))`, etc., for `Elem`. In a classical model semantics approach, we would expect `PARAM` to have a loose interpretation, accepting as models every algebra with a set of values for `Elem`, a constant value corresponding to `k`, and a binary operator corresponding to `bin`. The module `LIST` would then specify lists of `Elem`, for each of these possible models. Of course, with initial semantics, many usual "instances" of `LIST` (e.g., lists of `NATURAL`) would not be valid models of this specification which only accepts isomorphic models to the term algebra.

In the user manual for ASF+SDF [21] this kind of structuring with imports and renaming is proposed as a way to define "generic" specifications and their instances, as in the example 4.2 below.

Example 4.2 Given the specifications below for `SIMPLELIST` and `ELEM`:

```
module SIMPLELIST
  imports    ELEM
  exports
    sorts List
    context-free syntax
      nil               -> List
      cons (Elem, List) -> List


module ELEM
  exports
    sorts Elem
```

we can "instantiate" by renaming the sort Elem.

```
module ListNat
  imports    NATURAL
             SimpleList[Elem => Nat]
```

Of course, this proposal contradicts the idea of actual parameters being specializations the formal parameters as required in definition 2.1. Furthermore, only sort identifiers may be "instantiated" this way, and this is often not enough. In the list specification of example 4.1, for instance, this would not work, as the operators `k` and `bin` would not be instantiated by `zero` and `add`. It would only work if `PARAM` was purely composed of sorts.

Assume now that we have an operation capable of substituting `PARAM` by `NATURAL` and `Elem`, `k` and `bin` by `Nat`, `zero` and `add`. Through this operation we can obtain the specification of lists of natural numbers, with a sum operation computing the sum of all elements of a given list. We may also specify lists of boolean values, with `sum` giving the conjunction of all of the values of a list, through the substitution of `PARAM` by `BOOLEAN` and of `Elem`, `k` and `bin` by `Bool`, `true` and `and`, for instance. It is then possible to simulate classical parameterization and instantiation in the context of ELAN, and this is what we do in the FERUS tool.

We propose then to annotate the importation of a specification with a comment (e.g., `%% formal parameter`), stating that this import is supposed to represent a formal parameter for the module. This comment is ignored by regular ELAN tools as any other comment, but will be taken into account by FERUS. The instantiation operation provided by FERUS will then consider this annotated import as a formal parameter that may be instantiated. Once the user defines the desired actual parameter and instantiation mapping (or morphism), the instantiation operation will check if the conditions for the operation to be defined (def. 2.1) are satisfied and will carry out the transformation, generating a new ELAN specification, with the substitutions described above: substitution of the importation of the annotated formal parameter by the importation of the actual parameter (without any annotations this time), and substitution of all items of the formal parameter by the corresponding ones in the actual parameter. It is important to stress that all substitutions are actually carried out, generating a new specification module without any renamings (except possibly for previously existing ones). So, this approach may also be used for languages that do not provide the renaming construct.

Throughout the rest of the paper, we will then consider that an ELAN imported specification annotated with the `formal parameter` comment is a formal parameter in the classical sense, and that instantiation requires the specialization and push-out conditions as described in section 2. Conversely, generalization will aim at substituting regular imports of a specification by an annotated formal parameter.

5 Integration to the FERUS tool

The FERUS tool was designed to formally support the development of reusable algebraic specification components. This is accomplished through the use of *meta operations* capable of preserving algebraic properties. When such transformation operations are applied to a given component, FERUS generates a new component which has a well defined relationship with the one who has originated it. For instance the *renaming* operation available in FERUS changes sort and operator identifiers and generates isomorphic specifications (isomorphic signatures and the same class of models).

One may argue that a simple text editor would do this task well, but this approach is error prone, besides, FERUS operations provide more control over the properties of the components. This feature may be used in conjunction with other tools, as the development graphs [1] for CASL, facilitating "bookkeeping" activities related to reuse of a maximum of verifications of the original components in the newly generated ones. Furthermore, application conditions for the transformation operations may be quite complex, and in this case, tool support is extremely welcome.

The FERUS architecture, illustrated in figure 2, was conceived to be reusable, easily distributable, and maintainable. Considering these aspects, its components are organized as follows:

- *Internal Format Library*: library that implements an *internal graph format* representing the abstract syntax of an ELAN module. All operations are performed over this format, making them more efficient, since each graph

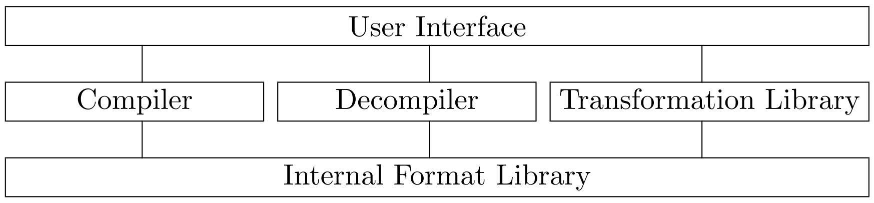

Fig. 2. FERUS Architectural Components

node represents uniquely a sort, operator, or variable declaration [16].

- *Transformation Library*: library that implements the transformations, making the operations easily available to be used and helping to separate user interface concerns and component manipulation.
- *Compiler*: interacts with some ELAN tools to obtain the parse tree of a module in ATerms format [20]. Then it is converted into the FERUS internal format.
- *Decompiler*: performs the often necessary task of retrieving the textual representation of a module stored in FERUS graph format.
- *User Interface*: integrates the above services with a wizard support (see figure 3).

Additionally, the tool disposes of functionalities like editing and executing (through ELAN environment) ELAN specification components.

The FERUS operations we've been referring to in this paper (instantiation, generalization and renaming) constitute the Transformation Library. Other available operations in this library are *extension*, an operation that adds new elements to the signature and/or axioms of a component; and *reduction*, that eliminates elements from the signature or from the axioms set. In the following, the three main operations in the context of this paper are presented through simple examples.

Renaming

Example 5.1 The result of the application of the FERUS renaming operation to the module `SIMPLELIST` of example 4.2, changing the sort identifier `List` to `List2` and the identifier of the operator `cons` to `cat` is:

```
module LIST2
  imports    Elem
  exports
    sorts List2
    context-free syntax
      nil                -> List2
      cat (Elem, List2) -> List2
```

The main differences between the FERUS rename operation and the ELAN and ASF+SDF built in renaming are shown in the table below.

Rename - FERUS	Renaming - ELAN
Renames sort and operator identifiers	Renames symbols, in our context, sort identifiers
Generates a new specification	The renaming is valid only in the context of the current specification
The generated specification is isomorphic with respect to the original one	No such property
Adapts the renamed module to a different context	Adapts the renamed imported modules to the context of the importing one
Applied to the local presentation (items introduced in the body of the specification)	Applied to imported specifications

Generalization

The *generalization* operation takes a component specifically developed for some context and makes it available to multiple uses in different contexts. In order to represent a larger class of models by a given specification, the generalization operation replaces imported specifications (usually with initial semantics) by a formal parameter from which the substituted specification is a specialization. In the case of ELAN, this formal parameter will be a `formal parameter` annotated import, as described in section 4. This operation is not standard. It was first proposed in [15] in the context of LPG and is one of the main particularities of FERUS. A more detailed description of this operation and of the issues concerning the definition of its arguments may be found e.g. in [17].

Example 5.2 Let's take again the list example, supposing now that we have the following module defining lists of natural numbers:

```
module LISTNAT
  imports   NATURAL
  exports
    sorts List
    context-free syntax
      nil             -> List
      cons (Nat, List) -> List
      sum  (List)      -> Nat
    variables
      "x" -> Nat
      "y" -> Nat
      "L" -> List
  rules
    [] sum(nil)        => zero
    [] sum(cons(x, L)) => add(x, sum(L))
```

It could be generalized over the sort `Nat`, substituting the import of `NATURAL` by the formal parameter `PARAM` of example 4.1, generating the generalized specification:

```
module LIST
  imports    PARAM              %% formal parameter
  exports
    sorts List
    context-free syntax
      nil                -> List
      cons (Elem, List)  -> List
      sum  (List)        -> Elem
    variables
      "x" -> Elem
      "y" -> Elem
      "L" -> List
  rules
    [] sum(nil)        => k
    [] sum(cons(x, L)) => bin(x, sum(L))
```

i.e., the last list specification of example 4.1 with the extra `formal parameter` annotation.

Instantiation

The reverse process, *instantiation*, is the substitution of the generic parameter by some more specialized specification, as described in section 2.

Example 5.3 The specification `LIST` of example 5.2 may be instantiated with the morphism from `PARAM` to `NATURAL` mapping `Elem`, `k` and `bin` into `Nat`, `zero` and `add`, respectively. The instantiation well-definedness conditions will be checked by FERUS, generating the specification `LISTNAT` of example 5.2. If we suppose that `NATURAL` also defines a multiplication operator `mult` and the constant `one`, for instance, we may also instantiate `LIST` into `LISTNAT2` (not shown, but very similar to `LISTNAT` with the substitution of all occurrences of `add` by `mult`), and now the operator `sum` would specify the product of all elements of the list[6].

[6] It would still be named `sum`. To make its name correspond to its new semantics, a further renaming operation is recommended.

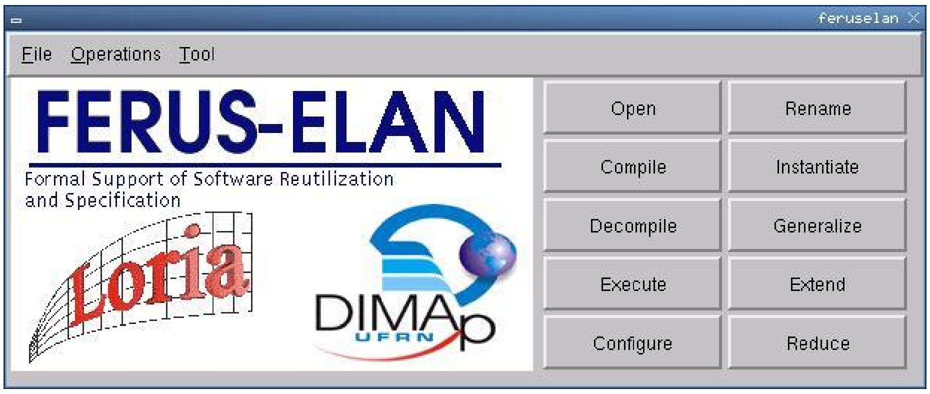

Fig. 3. FERUS user interface main window

6 Conclusions

We propose in this paper the use of meta transformation operations for algebraic specification components in order to improve modularization and simulate genericity in languages with initial semantics and restricted modularization constructs, as it is the case for ELAN and ASF+SDF. This approach has the merit of providing these extra-features in a behavioral way, without any changes to the language itself. Because these meta operations generate transformed specifications in the language, instead of being considered as new constructs of the language, there is no need to adapt any of the existing tools. In particular, all existing tools for these languages will simply ignore the extra comment included to indicate that a particular import should be regarded as a formal parameter.

This approach is being applied to the ELAN language through the use of the FERUS tool. This tool is being adapted from a previous version in development for the CASL algebraic specification language and provides a set of common algebraic specification transformation operations (renaming, instantiation, extension and reduction) together with a more original operation of generalization by parameterization. Genericity behavior is implemented via the pair of operations instantiate/generalize. The other operations provide some extra facilities in the evolutionary ELAN specification development.

Similar ideas may be found in the definition of Full Maude [11], and in the definition of the structuring constructs of CASL [8], with the main difference that in these works the language themselves are extended with modularity features. We claim that our approach is more easily applicable to any language due to the fact that we do not interfere with any existing tools for the language. Only the FERUS tool has to be adapted, and our current experience with the CASL and ELAN versions indicates that this can be done with reasonable effort.

Acknowledgements

The authors thank the members of the FERUS project, both on the brazilian side (DIMAp/UFRN) and on the french side (PROTHEO Team). They also thank Mark van den Brand and Jurgen Vinju, from CWI, Amsterdam, for their help with many implementation issues.

References

[1] Autexier, S. and T. Mossakowski, *Integrating HOL-CASL into the Development Graph Manager MAYA*, in: A. Armando, editor, *Proc. FroCoS'2002*, Lecture Notes in Artificial Intelligence **2309** (2002), pp. 2–17.

[2] Bergstra, J. A., J. Heering and P. Klint, "Algebraic specification," ACM Press, 1989.

[3] Bert, D., R. Echahed and J. Reynaud, *Reference manual of the LPG specification language and environment (release with disequations)*, Technical report, LGI-IMAG (1994), ftp available — site ftp.imag.fr.

[4] Borovanský, P., C. Kirchner, H. Kirchner, P.-E. Moreau and C. Ringeissen, *An Overview of* ELAN, in: C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science (1998).

[5] Borovanský, P., C. Kirchner, H. Kirchner and C. Ringeissen, *Rewriting with strategies in ELAN: a functional semantics*, International Journal of Foundations of Computer Science **12** (2001), pp. 69–98, also available as Technical Report A01-R-388, LORIA, Nancy (France).

[6] Cirstea, H. and C. Kirchner, *The rewriting calculus — Part I* and *II*, Logic Journal of the Interest Group in Pure and Applied Logics **9** (2001), pp. 427–498.

[7] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science (2001).

[8] CoFI, "The CoFI Algebraic Specification Language," (2003), available at the CoFI home page: http://www.brics.dk/Projects/CoFI.

[9] Dershowitz, N. and J.-P. Jouannaud, *Rewrite systems*, in: *Handbook of Theorectical Computer Science*, Elsevier Science Publishers B.V., 1990 .

[10] Deursen, A. v., *An overview of ASF+SDF*, in: A. v. Deursen, J. Heering and P. Klint, editors, *Language Prototyping: An Algebraic Specification Approach*, World Scientific Publishing Co., 1996 pp. 1–30.

[11] Durán, F., "Reflective Module Algebra with Applications to the Maude Language," Ph.D. thesis, University of Málaga (1999).

[12] Ehrig, H. and B. Mahr, "Fundamentals of algebraic specification 1 and 2," EATCS Monographs on Theoretical Computer Science **6 and 21**, Springer-Verlag, 1985/1990.

[13] Lima, G., A. M. Moreira, D. Déharbe, D. Pereira, D. Sena and J. Vidal, *FERUS: um ambiente de desenvolvimento de especificações CASL*, in: *Proceedings of SBES'2002 (Simpósio Brasileiro de Engenharia de Software): Sessão de ferramentas*, 2002, pp. 1–6.

[14] Martí Oliet, N. and J. Meseguer, *Rewriting logic: roadmap and bibliography*, Theoretical Computer Science **285** (2002), pp. 121–154.

[15] Moreira, A. M., "La Généralisation : un Outil pour la Réutilisation," Ph.D. thesis, INPG (1995).

[16] Moreira, A. M., C. Ringeissen, D. Déharbe and G. Lima, *Manipulating algebraic specifications with term-based and graph-based representations*, Journal of Algebraic and Logic Programming (2003), to appear.

[17] Moreira, A. M., C. Ringeissen and A. Santana, *A Tool Support for Reusing ELAN Rule-Based Components*, in: J. Giavitto and P. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming, RULE'03*, Eletronic Notes in Theoretical Computer Science **86.2**, 2003.

[18] Mossakowski, T., *Relating CASL with other specification languages: the institution level*, Theoretical Computer Science **286** (2002), pp. 367–475, guest editor: J.L. Fiadeiro.

[19] van den Brand, M. and J. Bergstra, *Syntax and semantics of a high-level intermediate representation for asf+sdf*, Tech. report, University of Amsterdam, Programming Research Group (1998).

[20] van den Brand, M., H. de Jong, P. Klint and P. Olivier, *Efficient annotated terms*, Software-Practice and Experience **30** (2000), pp. 259–291.

[21] van den Brand, M. and P. Klint, "ASF+SDF Meta-Environment User Manual," (2002), available at: http://www.cwi.nl/projects/MetaEnv.

[22] van den Brand, M., P.-E. Moreau and C. Ringeissen, *The ELAN environment: a rewriting logic environment based on ASF+SDF technology*, in: M. G. J. van den Brand and R. Lämmel, editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, Electronic Notes in Theoretical Computer Science **65**, Grenoble (France), 2002.

[23] van den Brand, M. and C. Ringeissen, *ASF+SDF Parsing Tools applied to ELAN*, in: K. Futatsugi, editor, *Proceedings of the third International Workshop on Rewriting Logic and Applications*, Electronic Notes in Theoretical Computer Science **36** (2000).

[24] Visser, E., "Syntax Definition for Language Prototyping." Ph.D. thesis, University of Amsterdam (1997).

[25] Wirsing, M., *Algebraic specification*, in: *Handbook of Theorectical Computer Science*, Elsevier Science Publishers B.V., 1990 .