A Tool Support for Reusing ELAN Rule-Based Components

Anamaria Martins Moreira, Christophe Ringeissen, Anderson Santana

A Tool Support for Reusing ELAN Rule-Based Components ⋆

Anamaria Martins Moreira [a,1], Christophe Ringeissen [b,2], Anderson Santana [a,3]

[a] *Universidade Federal do Rio Grande do Norte; Departamento de Informática e Matemática Aplicada; Campus Universitário; Lagoa Nova; 59072-970 Natal, RN; Brazil*

[b] *LORIA; Technopôle de Nancy-Brabois; Campus scientifique; 615, rue du Jardin Botanique; BP101; 54602 Villers-lès-Nancy Cedex; France*

Abstract

The adaptation of software components developed for a specific application in order to generate reusable components often includes some kind of generalization. This generalization may be carried out, for instance, by the renaming of some identifiers or by its parameterization. In our work, we are specially interested in the generalization by parameterization of algebraic specification components. Generalization and some other transformations on algebraic specifications are being integrated in the FERUS tool. This tool was initially developed for the Common Algebraic Specification Language called CASL, and we show in the paper its adaptation to the new version of the rule-based programming language ELAN.

1 Introduction

Formal specifications can provide significant support for software component reuse, as they allow tools to "understand" the semantics of the components they are manipulating. Tools that manipulate programming code can easily deal with syntactic features of components (e.g., renaming operations) but usually have a hard time when it comes to semantics. Formal specifications, with their "simpler" and precisely defined semantics and associated verification tools, contribute to the verification of the validity of semantic properties

⋆ This work is supported by a joint research project funded by CNPq and INRIA. It has been partly performed while the first and third authors were visiting LORIA.

[1] Email: anamaria@consiste.dimap.ufrn.br

[2] Email: Christophe.Ringeissen@loria.fr

[3] Email: anderson@consiste.dimap.ufrn.br

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of components in the different steps of the reuse process. For instance, they can be of great help on the generation of reusable components through the parameterization of more specific ones, supporting the process of creation of reusable components. So, we are interested in the area of programming *for* reuse. However, instead of creating components explicitly for a future reuse, we propose to create reusable components from an already developed application. Both choices have their pros and cons, and the second one presents as main advantage that the generated component has already been used at least once (in its original version), as already advocated e.g. by M. Wirsing in [13]. In this case, the main difficulty is to transform components that were developed for a particular application into effectively reusable components. This must be done through their generalization, and this is where our work fits.

In this paper, we propose a tool called FERUS, to support the reuse of components, via some operations, including one that aims at generalizing algebraic specification components by their parameterization. The main difficulty when trying to generalize by parameterization is to identify the "good" level of generalization for each component. Highly specific ones have small chances of being reused, but on the other hand, if a component is too general, its reuse will often be useless. It is necessary to state the semantic properties of a component that are considered "important" somehow (the component would loose its *raison d'être* if these properties were not satisfied). So, we need to be able to identify the requirements that a formal parameter should satisfy in order to preserve these stated properties in the generalization. When these stated properties are derived from equational axioms, a subset can be extracted from them and added to the formal parameter so that they are preserved in the generalized component. This simple technique provides sufficient conditions for the validity of the considered properties in the models of the more general specification, with the advantage of being easily computed by a simple algorithm.

The FERUS tool, which was firstly developed to deal with CASL [5] specifications, has the goal of supporting formal specification components development. It provides an environment for specification design and to build prototypes, that allows to edit, compile and execute specifications (given that the specification is executable). Its main feature is the possibility to derive new component through reuse driven transformation operations, for example, to create an instance of a parameterized module using the instantiate operation, and conversely to create a parameterized 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 [11]. The idea was to keep the tool architecture unchanged, thus only language specific features were subject of adaptation.

The paper is organized as follows: Section 2 aims at positioning the ELAN

rule-based programming language in the algebraic specification setting. The generalization (meta) operator is presented in Section 3 in a informal way. In the case we want to preserve a set of axioms, we present a simple algorithm to compute the signature morphism and the formal parameter required by the generalization operation. Generalization and other operations on algebraic specifications are implemented in the FERUS tool. In Section 4, we present a detailed view of the FERUS-ELAN instance and we discuss the adaptation issues when considering the ELAN rule-based programming language.

2 Algebraic Specifications and Genericity

Algebraic specifications [14] 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.

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. Recently, starting with Maude [4] and its underlying rewriting logic [6], on one hand, and ELAN and the ρ-calculus [3] 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 this paper, we consider the new version of ELAN called ELAN 4. This new version borrows the syntax of the ASF+SDF specification language [11], 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 [4]. 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 [10]. Structuring is mainly obtained through the importation and renaming of (parts of) imported specifications. Parameterization in these languages is very restricted, however, as only sorts are considered. Also, the semantics of a generic specification and of its instantiation are quite special, so we will discuss this in some more detail here.

In most algebraic specification languages, the parameter of a generic specification is supposed or required to present a loose semantics, accepting a large class of possible models. Then, instantiation is the substitution of this generic parameter by one of its "models" (a specification whose models are a subclass of the models of the parameter, possibly isomorphic, modulo signature translation). This *specialization condition* is needed for an instantiation to be defined. Because there is no loose interpretation of specifications in ELAN and ASF+SDF (only initial semantics), genericity cannot be treated in the same way. Parameterization and instantiation in these languages are just special cases of a more general construct, called *renaming* [12], and there is no requirement in the sense of specialization in the instantiation of a generic parameter. To be able to apply the theory of generalization to ELAN, we will interpret some ELAN specifications differently.

Example 2.1 Consider the specification of lists with an accumulation operation below where variable declarations have been omitted:

```
module List
  imports Psynt                        %% formal parameter
  exports
    sorts List
    context-free syntax
      nil                -> List
      cons (Elem, List)  -> List
      sum (List)         -> Elem
  rules
      [] sum(nil)        => k                        %% sum.nil
      [] sum(cons(x,l))  => bin(x, sum(l))           %% sum.cons

module Psynt
  exports
    sorts Elem
```

[4] To be precise, the current version of the FERUS tool does not cope with many other features of the language, as e.g., mixfix operators and priorities. This is due to the use of an incremental software development methodology, but these will be included in the following and have no influence in the presented work. This is not the case for strategies, which are much more than syntactic sugar and need special study.

```
context-free syntax
  k                  -> Elem
  bin (Elem, Elem)   -> Elem
```

In a classical model semantics approach, we would expect `Psynt` to have a loose interpretation, accepting as models every algebra with a set of values, a constant and a binary operator, and the module `List` would specify lists of `Elem`, for each of these possible models. On the other hand, with an initial semantics, we only have isomorphic models to the term algebra, with values `k`, `bin(k,k)`, `bin(k,bin(k,k))`, etc. Of course, with this kind of semantics, many usual instantiations of `List` (e.g., `List[Nat]`) would not be valid with the usual instantiation definition (the natural numbers are not isomorphic to this term algebra, and the specialization condition above does not hold). However, it is possible to *rename* `Elem`, `k` and `bin` into `Nat`, `0` and `+`, without any requirement concerning the models of these specifications. It is then possible to simulate classical parameterization and instantiation in the context of ELAN, and this is what we do in FERUS-ELAN. 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 above specialization condition. Conversely, generalization will aim at substituting regular imports of a specification by an annotated formal parameter.

3 Generalization

The generalization operation is the key for preparing a component to be kept in a library in order to be reused. Then, the instantiation may applied to reuse the generalized (generic) component. The main effect of the generalization operation is to safely substitute input specifications of a specification component by a formal parameter from which the substituted specification is a specialization. Roughly speaking, generalization corresponds to "enlarging" [5] the class of models over which we construct our specification, consequently "enlarging" the class of models of the new component. Conversely, instantiation reduces the class of models of the specification, and so reduces the class of models of the related component.

Let us now briefly present the generalization operation in a rather informal way. A detailed definition can be found in [7,8]. The generalization operator

Co' = `generalize` Co `via` *m* `with` PARAM

admits three arguments:

[5] We use here an intuitive notion of enlarging (generalization) or reducing (instantiation) a class of models, although the differences in the underlying signatures make the comparison not straightforward.

(i) A given specification Co which is built over some other imported specifications and may already be generic. In ELAN, it would have the form:
`module` Co `imports` *params-and-imports* `exports SPEC`
where `SPEC` consists of (local) sorts $Sort$(`SPEC`), operators Op(`SPEC`), variables Var(`SPEC`), axioms Ax(`SPEC`) which are said to be defined inside the body of Co. Note that local ingredients occurring in `SPEC` do not define neither a signature nor a specification due to the existence of *params* and *imports*, where we may for instance define a sort used in the profile of an operator in Op(`SPEC`), or an operator used in Ax(`SPEC`).

(ii) A specification Param, which is the potential parameter.

(iii) A signature morphism m from objects in Param to imported objects (including parameters, if any) of Co.

In order to be able to define the generalized specification Co', some conditions on Co, m and Param must be satisfied. Hence, the signature morphism m must be injective, so that it can be inverted, and it must correspond to a *specification morphism* from Param to the resulting imported specification *sp* of Co (i.e., the reduct or forgetful functor determined by m applied to models of *sp* must result in models of Param). In addition, we need some technical conditions (see [8] for details) to ensure that we get a syntactically consistent specification component without any references to removed objects. If all of these conditions hold, then we can substitute the previous imports and parameters of Co by the sole Param specification, renaming imported sorts and operators that appear in the local text of Co according to the inverse of m, thus leading to the following specification, using the `[...]` notation as a shortcut for the `%% formal parameter` annotation:

`module` Co' `[` Param `] exports` m_g(`SPEC`)

where the "translation" mapping m_g is a simple renaming of symbols derived from m^{-1}. It is applied throughout the body of the specification being generalized, renaming every occurrence of the generalized sorts and operators.

Example 3.1 Consider the ELAN specification below of lists of natural numbers with an operation `sum` that gives the total sum of all elements of a given list. It could be generalized into the previous generic list specification `List` through:

```
List = generalize List_Nat via [Elem |-> Nat, k |-> 0, bin |-> +]
                           with Psynt

module List_Nat
 imports Nat
 exports
   sorts List
   context-free syntax
      nil                        -> List
```

```
      cons (Nat, List)          -> List
      sum (List)                -> Nat
  rules
      [] sum(nil)               => 0            %% sum.nil
      [] sum(cons(x,l))         => x + sum(l)   %% sum.cons

module Nat
 exports
   sorts Nat
   context-free syntax
      0                        -> Nat
      suc(Nat)                 -> Nat
      Nat ``+'' Nat            -> Nat
  rules
      [] x + 0                  => x            %% add.0
      [] x + suc(y)             => suc(x+y)     %% add.suc
```

Note that it may be important to rename locally declared sorts and operators after generalization. For instance, in this example, the sort for lists of naturals could have been named `ListNat`, and in this case, one would like the corresponding sort of the generalized component to be named `List`. This could be done through a renaming operation in ELAN.

The generalization operator presented above is basically a tool for, once the desired generalization is defined (by the morphism m and the specification PARAM), safely executing the corresponding transformation of the component. The most delicate part in the process is however the definition of the desired generalization, i.e., to define m and PARAM.

To provide some support on this step, we provide an algorithm that computes a specification PARAM for a given set of sorts we want to generalize. This algorithm finds which operators have to occur in the formal parameter PARAM. Furthermore, provided that we want to preserve a set of axioms in the generalized component, this algorithm selects which axioms have to be added to the formal parameter PARAM. When the set of axioms is empty (resp. non-empty), we are talking about syntactic (resp. semantic) generalization. The general form of generalization is particularly interesting when we want for instance to preserve the proof of a theorem in the generalized component: in that case, the set of axioms corresponds to all axioms involved in the proof.

Obtaining the Generalization Morphism

We present the algorithm computing the arguments needed by the generalization operator. To present it, we use a little more notation:

- Given an axiom e, $operators\_of(e)$ returns the set of operators occurring in e, and by extension to sets of axioms, we define $operators\_of(\{e\}_{e \in E}) =$

$\{operators\_of(e)\}_{e \in E}$.

- Given a operator o, $sorts\_of(o)$ returns the set of sorts occurring in the profile of o, and by extension to sets of operators, we define $sorts\_of(\{o\}_{o \in O}) = \{sorts\_of(o)\}_{o \in O}$. Moreover, given a set of axioms E, we define $sorts\_of(E) = sorts\_of(operators\_of(E))$.

Input: Let GS be a set of sorts to generalize in a component Co, and E be a set of axioms in Co we want to preserve by generalization. Include to E all trivial axioms $f(\boldsymbol{x}) = f(\boldsymbol{x})$ for each operator f occurring in the component Co.

1 `repeat`
2 $\quad E^g := \{e \in E - Ax(\texttt{SPEC}) | sorts\_of(e) \cap GS \neq \emptyset\}$
3 $\quad Pend := (sorts\_of(E^g) - GS)$
4 $\quad GS := GS \cup Pend$
5 `until` $Pend = \emptyset$

Output: The generalization is defined for the morphism

$$m_{\texttt{SEM}} : \texttt{PSEM}[\texttt{sorts } s'_1, .., s'_m \texttt{ ops } op'_1, .., op'_j] \rightarrow sp[\texttt{sorts } s_1, .., s_m \texttt{ ops } op_1, .., op_j]$$

where

- sp is the whole imported and/or parameter specification of Co,
- the objects of the specification `PSEM` are the sorts $GS = \{s_1, \ldots, s_m\}$ and operators in $operators\_of(E^g) = \{op_1, \ldots, op_j\}$ with respective new names $\{s'_1, \ldots, s'_m\}$ and $\{op'_1, \ldots, op'_j\}$, and the non-trivial axioms of E^g with the corresponding renamings.

Example 3.2 Consider again the ELAN specification of lists of natural numbers given in Example 3.1. The equation `th1` below is one of its theorems as shows the rewrite proof $p1$.

```
sum(cons(X, nil)) = X                    %% th1
```

$$p1 = \begin{cases} sum(cons(X, nil)) & \rightarrow_{sum.cons} \\ X + sum(nil) & \rightarrow_{sum.nil} \\ X + 0 & \rightarrow_{add.0} \\ X & \end{cases}$$

The generalization of the sort `Nat` with preservation of (axioms in) $p1$ gives:

```
List = generalize List_Nat via [Elem |-> Nat, k |-> 0, bin |-> +]
                          with Psem

module List
 imports Psem
```

```
 exports
  ... (same as before, but now with th1 as consequence)

module Psem
  imports Psynt
  rules
      [] bin(X,k) => X                %% bin.k
```

4 FERUS-ELAN

In its current development stage, FERUS-ELAN is mainly intended to support transformation operations over specification components written in the ELAN language. Furthermore, common specification development functionalities (edition, compilation/decompilation, and execution) are provided. The specification transformation operations currently available in FERUS are described below. It should be pointed out that these are meta-operations. Most of the FERUS operations are strongly related to some of CASL structuring constructs, this is mainly because they are quite standard in the algebraic specification domain, but we do not try to give them exactly the same semantics. Here, the idea is to verify applicability conditions for a given transformation operation and carry it out, whenever possible, generating a new ELAN specification component. One interesting consequence of this approach is the potential adaptability of the FERUS tool to specification languages different from CASL, as shown by the presented FERUS-ELAN instance, even if they do not include the same sophisticated structuring constructs as CASL.

The available transformation operations are:

rename — allows the substitution of sort and operator identifiers by new ones, provided that these renamings preserve the semantics of the specification component being renamed (isomorphic models).

extend — allows to extend a specification component by the addition of sorts, operations and/or axioms.

reduce — allows to eliminate or hide parts of a specification component.

instantiate — allows the substitution of formal parameters of a generic specification component by actual parameter specifications (specifications such that there is a morphism from the formal parameter to them).

generalize — allows the substitution of imported or extended specifications of a specification component by a formal parameter from which the substituted specification is a specialization (there exists a morphism from the added formal parameter to the removed imported or extended specification).

For efficiency reasons, these transformation operations are implemented in a graph representation of the specifications, detailed in Section 4.2.

In addition to the above transformation operations, FERUS-ELAN provides some functionalities to interact with ELAN:

compile — parses and checks a given specification component and generates its representation in a graph-like internal format.

decompile — generates ELAN text from the internal representation of a specification. It is particularly useful to visualize the results of a transformation operation.

Both functionalities are implemented using the ATerm exchange format to interact with external tools of ELAN (version 4).

A general picture of the FERUS-ELAN tool is given in the next section.

4.1 General Architecture of FERUS-ELAN

The FERUS-ELAN design is identical to the version dedicated to CASL. This reinforces that the tool modeling is appropriate for the domain of algebraic specification languages. The architectural components of FERUS-ELAN illustrated in figure 1 are described as follows :

compiler — its task is to translate specification components written in a sublanguage of ELAN to the FERUS internal format. The compiler uses a set of libraries provided by ELAN to catch the parse trees of each specification component, which is an important change with respect to the FERUS-CASL version that works with abstract syntax trees. Once these parse trees are identified and analyzed, the compiler interacts with libelan in order to create and store the graph nodes that represent the specification in a repository of compiled modules.

library implementing the internal format — libelan disposes of a set of useful functions to create and to handle the data structure that represents the abstract syntax of elan as graphs. In fact this library was first implemented for a model-checker and was also been successfully applied for the FERUS-CASL instance (libcasl). This means that the graph-like data structure is kept the same, only the organization of nodes had to change in order to represent the abstract syntax of each different language.

library implementing the transformation operations — libferuselan implements the operations like renaming, generalization, among others. Some of the operations could be implemented with the same semantic for both CASL and ELAN, but since parameterization mechanisms differ it was necessary a to do a study of how to achieve generalization in ELAN. The goal of libferuselan is to keep implementation details hidden and let the tool's architecture open to be reused by other tool developers.

graphical user interface — intended to support the user in the process of component reuse. Indeed, the verification of some of the correctness

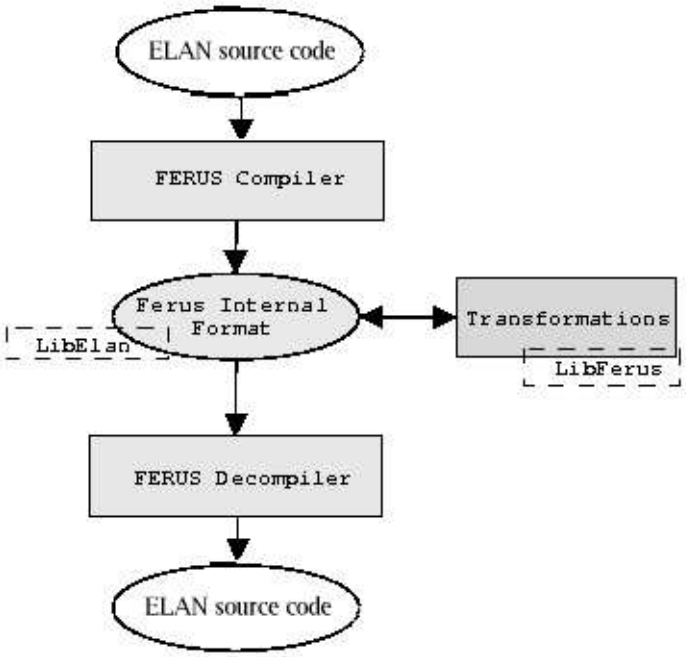

Fig. 1. FERUS-ELAN architecture

preconditions of transformation operations is not trivial and the user may need some support in the definition of their parameters. Therefore, we developed special-purpose user-interfaces, based on the wizard metaphor [2].

decompiler — retrieves the textual representation of a component contained in the FERUS-ELAN repository is ELAN sublanguage dealt by the tool. It is basically the inverse of compilation, so it also uses the functionalities contained in libelan, but this time it reads from the repository, and after visiting each graph node it outputs the corresponding source code in ELAN.

4.2 libelan: FERUS-ELAN Internal Format

Both versions of FERUS use an graph format to represent the abstract syntax of specifications. This format is more suitable for the transformations than using a maximal sub term sharing format as ATerms [1]. A discussion concerning the pros and cons of each approach can be found in [9].

libelan implements the functionalities needed to handle this format in memory as well as its storage in a repository of compiled specifications. It is important to stress that although it is not the same instance used to represent CASL specifications, the format itself is very similar to the one for libcasl, presented in [9]. Only the set of nodes has been changed, so that it could represent the abstract syntax for ELAN modules.

```
/* elanKind - Enumeration structure defining the kind of nodes
 * needed for a complete definition of the specifications.
 */
typedef enum {
  elanModuleDef, //Module definition
  elanImport, //Imported modules
  elanDefSection, //Exports section definition
  elanSortDef, //Sort declaration
  elanOperatorDef, //Operator declaration
  elanVariableDef, //Variable Declaration
  elanAxiomDef,  //Axiom
  elanVariableInst,  //Occurrence of variable
  elanOperation,  //
  elanList //General list
} elanKind;
```

Fig. 2. Definition of the node kinds in libelan

```
typedef union elanNodeRec_ {
  ...
  struct {
    elanRef position;           //elanNode position
    void * aToolInfo;           // free slot (tool-specific)
    elanKind aKind;             //elanModuleDef - kind of the node
    elanRef nRoot;              //Root specification
    unsigned aLine;             //Line in the source code
    char * aIdentifier;         //module's Identifier
    elanRef nImports;           /* list of elanImport */
    elanRef nExports;           /* Module's Exports section */
    elanRef nHiddens;           /* Module's Hiddens section */
    elanRef nAxiomDefs;         /* list of AxiomDef */
  } elanModuleDef;  //Module definition
 ...
} elanNodeRec;
```

Fig. 3. Structure of a specification definition node (kind `elanModuleDef`).

4.2.1 Implementation details for libelan

This section contains excerpts of the source code (in C) from libelan.

Figure 2 contains the definition of the data type `elanKind` that enumerates the different kinds of nodes employed to represent ELAN specifications. For each value of `elanKind`, the library contains a record type definition, as that given in Figure 3 for specification definitions.

Note that all the structure definitions are grouped in a single data type `elanNodeRec`. Direct access to these structures is not provided to the user, and only the data type `elanNode`, defined as a pointer to `elanNodeRec`, shall

```
extern elanNode elanMakeModuleDef
(int paLine,
 char * paIdentifier,
 elanNode pnImports,
 elanNode pnExports,
 elanNode pnHiddens,
 elanNode pnAxiomDefs
 );
```

Fig. 4. Example of declaration for a constructor.

be manipulated by client code, using a set of routines defined in the interface of the library.

The attributes `aKind` and `aToolInfo` and the edge labeled `nRoot` are common to all the different node signatures. `aKind`, obviously, is the kind of the node, `aToolInfo` is a slot where client applications may store specific data as a pointer. The `nRoot` edge links the node to the root of the subgraph of the specification it belongs to.

In the internal state of the library, each node is uniquely referenced by a pair composed of the identifier of the specification it belongs to, and its position in the node table. libelan has internal routines that, given a node reference, returns the actual node and vice-versa.

libelan interface provides initialization routine, constructors, and accessors routines. For instance, Figure 4 presents the constructor for specification definition nodes. Additionally, libelan provides file input and output routines, and thus may be used to maintain specification libraries as well as a mean to communicate between tools.

4.3 libferuselan: Transformations

This library was designed to make transformations available to the tool and to allow separation of concerns between the user interface support and the transformation functionalities.

4.3.1 Implementation details for libferuselan

Essentially a transformation generates a new component given some parameters. This can be seen as an abstraction in the following form:

```
NewComponent = Operation OldComponent with Parameters
```

Hence, for the generalization transformation, `Parameters` consists of a `Param` module and a mapping to describe the signature morphism, as shown in Section 3.

The libferuselan library was intended to implement this kind of abstraction, and to make it transparent to the programmer, whose concerns should be how to build the correct mappings accordingly to each transformation listed in Figure 5. Once the mapping is done, all he has to do is to apply the mapping

```
typedef enum {
  elanRenaming,
  elanExtension,
  elanReduction,
  elanInstantiation,
  elanGeneralization
} elanTransformationKind;
```

Fig. 5. Enumeration of kinds of transformations that can be applied to ELAN modules.

```
extern int applyMapping (
  elanNode pModule,
  const char * pNewModuleIdentifier,
  elanMapping pMapping
);
```

Fig. 6. Transformation application function.

be calling a single function (see Figure 6), that makes the interface with the library similar to the form presented above.

5 Conclusion

This paper reports the adaptation of the FERUS tool for the new ELAN language (version 4). This has led to a new version called FERUS-ELAN, which is already able to deal with a significant subset of the ELAN language. Our experiments demonstrate the efficiency of its internal format over which we apply operations for reusability of algebraic specification components. In the future, we will extend the current implementation in order to support the complete ELAN language.

For a specification language like ELAN, it is important to have a good tool support, possibly by reusing and adapting existing tools developed for related languages. In this direction, the main advance has been obtained with the adaptation of the metaEnvironment initially developed for ASF+SDF [11]. This metaEnvironment is component-based, and the interoperability of components is supported by a Toolbus - a software coordination architecture. As a tool for reuse-driven transformations of ELAN specifications, we envision to connect FERUS-ELAN to the Toolbus and so to integrate our tool to the new ELAN metaEnvironment. For a better integration, we must ease the execution (compilation/interpretation) of ELAN specifications generated by FERUS-ELAN. Obviously, the decompiler can be used to obtain the textual representation of an ELAN specification, but this well-formed specification has to be parsed again. Provided that an abstract representation is used as an exchange format, we could imagine to decompile the safe results generated by FERUS-ELAN and to give them directly to ELAN execution tools, without

introducing a non-necessary parsing phase. Thus, the problem of executing specifications generated by FERUS-ELAN illustrates the interest of using a common exchange format for the connection of ELAN-related tools.

References

[1] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[2] J. Carroll, R. Mack, and W. Kellogg. Interface metaphors and user interface design. In *Handbook of Human-Computer Interaction*. Elsevier, 1988.

[3] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[4] F. Durán. *Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[5] CoFI group. *The CoFI Algebraic Specification Language*. Available at the CoFI home page: http://www.briks.dk/Projects/CoFI.

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

[7] A. Martins. *La Généralisation : un Outil pour la Réutilisation*. PhD thesis, INPG, March 1995.

[8] A. Martins and C. Ringeissen. Generalizing CASL specification components and preserving rewrite proofs. Technical report, INRIA, 2003. to be published.

[9] Anamaria Martins Moreira, Christophe Ringeissen, David Déharbe, and Gleydson Lima. Manipulating Algebraic Specifications with Term-based and Graph-based Representations. submitted to Journal of Algebraic and Logic Programming, special issue on ATerms, 2003.

[10] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, September 2002. Guest editor: J.L. Fiadeiro.

[11] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen. The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65, 2002.

[12] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sep. 1997.

[13] M. Wirsing. Algebraic description of reusable software components. Technical Report MIP-8816, Fakultät für Mathematik und Informatik - Universität Passau, 1988.

[14] M. Wirsing. Algebraic specification. In *Handbook of Theorectical Computer Science*, chapter 13. Elsevier Science Publishers B.V., 1990.