Generalizing CASL Specification Components and Preserving Rewrite Proofs

Anamaria Martins, Christophe Ringeissen

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Generalizing CASL Specification Components and Preserving Rewrite Proofs

Anamaria Martins Moreira, Christophe Ringeissen

N° 4938

Septembre 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4938--FR+ENG

Generalizing CASL Specification Components and Preserving Rewrite Proofs*

Anamaria Martins Moreira†, Christophe Ringeissen‡

Thème 2 — Génie logiciel
et calcul symbolique
Projet PROTHEO

Rapport de recherche n° 4938 — Septembre 2003 — 34 pages

Abstract: We propose the theoretical basis of a tool for the generation of reusable CASL specification components by generalization of existing ones. The underlying idea is, given a component and a set of semantic properties that it satisfies and that we want to preserve, to find a parameterized, more general, component satisfying the following conditions: the original component is one of its possible instantiations, and any of its instantiations satisfy the stated properties. We present here both the definition of the generalization operation for CASL and the problem of preserving properties in the generalized component. To guarantee the preservation of properties, we propose to preserve their proofs, concentrating on the use of rewrite proofs. This technique provides a simple way to find sufficient conditions for the preservation of the corresponding properties. This work is being integrated in the specification component development tool FERUS, under development for the CASL language, using ELAN as the rewrite proof engine.

Key-words: algebraic specifications, rewriting systems, component parameterization, proof generalization

* This work is supported by a joint research project funded by CNPq and INRIA.

† Universidade Federal do Rio Grande do Norte — UFRN, Natal, RN, Brazil. Email: anamaria@dimap.ufrn.br

‡ Christophe.Ringeissen@loria.fr

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Généralisation de composants de spécification CASL et préservation de preuves par réécriture

Résumé : On propose les fondements théoriques d'un outil permettant d'engendrer des composants réutilisables de spécification CASL par la généralisation de composants existants. Etant donné un composant et un ensemble de propriétés sémantiques qu'il satisfait et qu'on veut préserver, l'idée sous-jacente consiste à trouver un composant paramétré, plus général, satisfaisant les propriétés suivantes: le composant original doit être une de ses instances possibles, et chacune de ses instances doit satisfaire les propriétés requises. On présente dans ce papier à la fois la définition de l'opérateur de généralisation pour CASL et le problème de la préservation de propriétés dans le composant généralisé. Pour garantir la préservation de propriétés, on propose par exemple de préserver leurs preuves, en s'intéressant plus particulièrement aux preuves par réécriture. La technique proposée fournit une façon simple de trouver des conditions suffisantes pour assurer la préservation des propriétés correspondantes. Ce travail est en cours d'intégration dans l'outil de développement de composants appelé FERUS, en développement pour le language CASL, utilisant le moteur d'ELAN pour construire les preuves par réécriture.

Mots-clés : spécifications algébriques, systèmes de réécriture, paramétrisation de composants, généralisation de preuve

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 programing 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 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 and maintenance of libraries of reusable components.

Among specification formalisms, algebraic specifications are well suited for our work, being adequate to the specification of components and given their strong links with the object oriented programming paradigm, largely associated with reuse in the literature. Additionally, the genericity mechanism encountered in most algebraic specification languages provides great flexibility in the desired abstraction level and the means to require semantic properties of potential instantiation parameters. Model-based specification languages like Z [22] or B [1] lack precisely this ability to parameterize specifications by other specifications, presenting only limited facilities for imposing restrictions on parameters. As we show in this paper, this is a major drawback concerning the generalization of specifications.

In this paper, we propose to generalize (CASL [1][12]) 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). With this information, we propose means to identify the requirements that a formal parameter should satisfy in order to preserve these stated properties in the generalization. To reach this goal, we consider proofs for these properties in the original (non-generalized) context and identify the conditions under which these proofs are reproducible in the generalized context. When these known proofs are rewrite proofs, a set of equational axioms can be extracted from them and added to the formal parameter so that they are preserved in the process. 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. This work is being integrated in the FERUS tool, under development as part of a Franco-Brazilian cooperation project, with the goal of providing support to

[1] Common Algebraic Specification Language, developed by the CoFI (Common Framework Initiative) group.

the process of creating and maintaining libraries of reusable algebraic specification components. It deals with the sublanguage of CASL described in Section 2 and interfaces with the rewrite engine of ELAN [7] in order to execute equational specifications and to prove equational theorems.

Briefly, this paper contributes to 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 advocated e.g. by M. Wirsing in [24]. 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.

This paper is structured as follows: Section 2 aims at defining the sublanguage of CASL we are interested in. The generalization (meta) operator is presented in Section 3. We first introduce some definitions that help us to state formally the result of the generalization operator, in the context of CASL. The problem of defining the desired generalization is addressed in Section 4. First, we consider the notion of *syntactic generalization*, where only sorts and operators are generalized. Then, we introduce the problem of generalizing *semantic properties*, that is, equational theorems. In Section 5, we show how to preserve rewrite proofs of equational theorems by defining the adequate parameter needed to build the generalized component. The implementation issue of the generalization process is discussed in Section 6, where we present how to use different tools developed for CASL. Before concluding, we show in Section 7 a possible technique to generalize a theorem for which there is no rewrite proof. Section 8 concludes with final remarks and future works.

2 CASL

The last decades have seen a proliferation of algebraic specification languages. Even if all these languages have some evident similarities, each of them has its own specificity, and a lot of different basic algebraic specifications concepts are used. This diversity appears to be a significant obstacle to the dissemination of algebraic specification methods in the education and to make them usable by the industry.

The Common Framework Initiative (CoFI) started several years ago in order to reach an agreement on a common framework for algebraic specification and development. One of the initial goals of CoFI was to promote a common specification language with uniform, user-friendly syntax and straightforward semantics, together with good documentation and tool support. The specification language developed by CoFI is called CASL [3], the Common Algebraic Specification Language. CASL is indeed the heart of a family of languages. Some specific tools will focus on well-defined *sub-languages* of CASL obtained by syntactic or semantics restrictions, while *extensions* of CASL are defined to support various paradigms and applications. For

example, there is an extension of CASL to support a form of partial higher order logic [21].

The design of CASL is now finished [13], a complete formal semantics has been given [12], and several tools are being implemented [18]. For developing CASL tools, the idea was to *reuse* the available parsing technology [23] and existing tools like theorems provers [20, 18, 4] and rewrite engines [15].

The FERUS tool aims at providing support to the process of creating and maintaining reusable algebraic specification components. In its first version, this tool will be working on a sublanguage of CASL. This sublanguage has not been submitted to approval to CoFI, but it is close to classical simple algebraic specification languages such as ACT ONE [10] and Larch [14]. It is based on conditional equational logic, with many-sorted total operators, the built-in equality predicate as the unique predicate and the free data-type construction as the abstract data type specification mechanism. Two kinds of algebraic specifications can be distinguished in our CASL sublanguage:

- *basic specifications:* we consider declarations of many-sorted total operators, and conditional equational axioms. As an example, let us consider a very simple specification of *Naturals*, named NAT, which involves a free type construct for the sort *nat*, and the declaration of *add* and *max*, together with the axioms defining these operators over *nat*. Note that all axioms are named using the annotation %(...)%.

  spec NAT =
  free type $nat ::= zero \mid succ(nat)$
  ops
  $add : \ nat \times nat \to nat;$
  $max : \ nat \times nat \to nat;$
  vars $x, y : nat$
  - $add(x, zero) = x$ %(add.0)%
  - $add(x, succ(y)) = succ(add(x, y))$ %(add.succ)%
  - $max(zero, x) = x$ %(max.0l)%
  - $max(x, zero) = x$ %(max.0r)%
  - $max(succ(x), succ(y)) = succ(max(x, y))$ %(max.succ)%

  The **free type** construct states that the carrier set of the corresponding sort is freely generated from the constructors (*no junk* and *no confusion*). If the other declared operators are completely and consistently defined, we have that the models of this kind of specification is the class of initial models. It is also possible to specify larger classes of models with a loose semantics, considering all models of a given signature and set of axioms. This is the case in the COMMUTATIVITY specification below, which does not contain the free type constraint and is satisfied by any data-type with a commutative binary operator.

spec COMMUTATIVITY =
  sort *elem*
  op $bin : elem \times elem \rightarrow elem$
  vars $x, y : elem$
  • $bin(x, y) = bin(y, x)$ %(com)%

- *structured specifications:* we consider unions, extensions and generic specifications. Genericity of specifications is made explicit using (formal) parameters. Also, we only work with named specifications, i.e, closed, in the context of the CASL language [12]. It is important to note that parameters are arbitrary specifications. This feature will be used in our framework to require explicitly that parameters satisfy specific sets of axioms.

  For instance, we give below the generic specification of *Lists* of elements that may be compared through an equivalence operator, the specification that characterizes the properties of this operator (reflexivity, symmetry and transitivity), and how the generic specification can be instantiated by the specification of *Naturals*, in order to build the specification of *Lists of Naturals*. This will give the reader an intuition on how genericity and the instantiation of parameters works in CASL. BOOL, not shown, can be assumed to be a specification of booleans with the usual boolean operators.

spec EQUIV = BOOL **then**
  sort *elem*
  op $eq : elem \times elem \rightarrow bool$
  vars $x, y, z : elem$
  • $eq(x, x) = T$ %(reflexivity)%
  • $eq(x, y) = eq(y, x)$ %(symmetry)%
  • $b\_imp(b\_and(eq(x, y), eq(y, z)), eq(x, z)) = T$ %(transitivity)%

spec LIST[EQUIV] =
  free type $list ::= nil \mid cons(elem; list)$
  op $eq : list \times list \rightarrow bool$
  vars
    $x, x1, x2 : elem$;
    $l, l1, l2 : list$
  • $eq(nil, nil) = T$ %(eq_nils)%
  • $eq(nil, cons(x, l)) = F$ %(neq_nilcons)%
  • $eq(cons(x, l), nil) = F$ %(neq_consnil)%
  • $eq(cons(x1, l1), cons(x2, l2)) = b\_and(eq(x1, x2), eq(l1, l2))$ %(eq_cons)%

The models of the above specification are freely generated from each possible model of specifications EQUIV. The instantiation of the generic specification LIST[] with NATEQ (NAT extended with a equivalence operator *eq* defined as usual), corresponds to one of these models, modulo signature differences, and is achieved by using a (uniquely defined) fitting of the parameter symbols to the argument symbols:

spec LIST_NAT = LIST [NATEQ **fit** *elem* $\mapsto$ *nat*, *eq* $\mapsto$ *eq*]

In the terminology used by Mossakowski in [19], the above sublanguage for basic specifications corresponds to an extension of $CCond^{=}$ (*conditional equational logic with equality and sort generation constraints*), which may be called $CNCond^{=}$, where atoms in the conclusions of conditional axioms may be negated[2]. With these restrictions, and those of the structured specification level, we obtain a language that has some good properties in terms of existence of initial models and free functors (liberality), compatibility with the theory developed in [16] for the LPG language [5], and for which rewriting techniques can be applied using, for instance, rule-based systems like ELAN [7]. It is complete enough to serve as a basis for our proposal and simple enough not to hinder the applicability of the tool in the details of a more complex language.

3 Generalization for CASL

The generalization operation by parameterization is dual to the instantiation operation presented in the previous section and the key for preparing a component to be kept in a library in order to be reused (instantiation may then be applied to reuse it). Note that, unlikely the instantiation cited above, one of the structuring constructs of the CASL language, this is a meta operator. In the FERUS tool, regular CASL structuring constructs will also be implemented as meta-operators. They have the goal of computing a kind of normalization of a CASL specification, simplifying its structure, when the operation is well defined, i.e., when it generates a well formed specification from a well formed specification. In this paper, we concentrate on the generalization operator.

The generalization operator that we propose has as main effect the safe substitution of input specifications of a specification component by a formal parameter from which the substituted specification is a specialization. This corresponds to "enlarging" [3] the class of models over which we construct our specification, consequently "enlarging" the class of models of the new component.

[2] This feature is needed to express the disjointness of constructors, associated with the free type construct.

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

Example 3.1 *Consider the specification below of binary trees of natural numbers, with an operation sum that gives the sum of all node values of a given tree, and where the specification* NAT *is the one presented in section 2.*

spec NATTREE = NAT ***then***
  free type $nattree ::= empty \mid bin(nattree; nat; nattree)$
  op $sum : nattree \to nat$
  vars
    $a1, a2 : nattree;$
    $tn : nat;$
  - $sum(empty) = zero;$ %(sum.empty)%
  - $sum(bin(a1, tn, a2)) = add(tn, add(sum(a1), sum(a2)))$ %(sum.bin)%

This specification may be generalized[4] *with our generalization operator into*

spec BTREE [PARAM1] =
  free type $nattree ::= empty \mid bin(nattree; s; nattree)$
  op $sum : nattree \to s$
  vars
    $a1, a2 : nattree;$
    $tn : s;$
  - $sum(empty) = k;$ %(sum.empty)%
  - $sum(bin(a1, tn, a2)) = bop(tn, bop(sum(a1), sum(a2)))$ %(sum.bin)%

where

spec PARAM1 =
  sort s
  ops
    k: s;
    *bop: s * s -> s*
  end

Actually, in CASL, there is no restriction on the semantic interpretation of parameter and imported specifications. It is expected however that parameters have a loose semantics while imports correspond more often to data types. For simplicity, we assume this practice in the motivation and informal presentation of the generalization operation, but the definitions presented here do not rely on it and work in more general situations.

We now formally define this meta generalization operator, presenting the conditions that must be satisfied for the operation to be carried out safely (i.e., generating a well formed specification component) and the obtained results. Definitions 3.1 and

[4]Note that the local identifiers (e.g., *nattree* and *sum*) are not automatically renamed, and their old names may not reflect their new generalized semantics, as it is the case here. This means that a generalization will often be followed by a renaming of localy defined identifiers.

3.2 introduce conditions for the generalization operation to be defined, while definitions 3.3 and 3.4 indicate how to obtain the generalized specification component. In all of these definitions and throughout the paper we consider the given original specification to be

spec Co [*params*] **given** *imports* = SPEC **end**

or, if it is not already parameterized,

spec Co = *imports* **then** SPEC **end**

The resulting signature of all imports and formal parameters is denoted Σ, and $\Delta : \Sigma \hookrightarrow \Sigma'$, is the extension of this signature by the sorts and operators locally declared inside SPEC $= (SI', OpI', VI', FD', AxI')$, where SI', OpI', VI' are respectively the sets of sorts, operators and variables declared in SPEC; AxI' is the set of axioms introduced by SPEC; and FD' corresponds to the free datatype declarations of SPEC[5]. The complete signature corresponding to Co is denoted by Σ', where $\Sigma' = (S', Op')$. Also, the following functions are used in these definitions:

- 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 $sorts\_of(E) = sorts\_of(operators\_of(E))$.
- Given a component Co, $ImpOps$(Co) stands for the set of operators of the resulting imports and parameters signature of the specification component (Σ, for the above Co) that occur in any of its local axioms (AxI' for Co). $LocalOps$(Co) stands for the set of locally declared operators (either in OpI' or FD', in the case of Co).

This preamble will not be repeated in each of the definitions, but should be assumed by the reader.

Definition 3.1 (dangling operators condition) *Given a specification* Co *as described above, a signature morphism* $m : \Sigma_{gp} \to \Sigma$ *is said to satisfy the* dangling operators condition *for* Co *if*

$$\forall s \in m(\Sigma_{gp}), \forall op \in ImpOps(\text{Co}), s \in sorts\_of(op) \Rightarrow op \in m(\Sigma_{gp})$$

[5]Note that free datatype declarations can be decomposed into sort declarations, operator declarations and axioms. So, the rules and definitions that apply to these objects can be applied to each of its components.

This condition aims to guarantee that the axioms of SPEC are well formed after generalization. For instance, if an imported sort *nat* is generalized and substituted by s and the natural addition occurs in an axiom, it must also be generalized and substituted by a corresponding binary operation over s.

Example 3.2 *Consider the specification of binary trees of natural numbers of example 3.1.*

The morphism

$$m : [\textbf{\textit{sorts}}\ s\ \textbf{\textit{ops}}\ k : s, unop : s \to s, bop : s * s \to s] \to \\ [\textbf{\textit{sorts}}\ nat\ \textbf{\textit{ops}}\ zero : nat, suc : nat \to nat, add : nat * nat \to nat]$$

satisfies the dangling operators condition, because the only NAT *operator that is not in the image of m is max, and it is not used in the axioms of* NATTREE. *This morphism is not minimum, in the sense that suc is attained by m, but not used in* NATTREE. *On the other hand, the morphism*

$$m1 : [\textbf{\textit{sorts}}\ s] \to [\textbf{\textit{sorts}}\ nat]$$

does not satisfy the dangling operators condition, as it does not attain $zero$ and add (operators on nat that are used in the local axioms).

Definition 3.2 (generalization condition) *Given a specification* CO *as described above, a signature morphism $m : \Sigma_{gp} \to \Sigma$ is said to satisfy the* generalization condition *for* CO *if*

$$\forall s \in \Sigma, s \in sorts\_of(LocalOps(\text{CO}) \cup ImpOps(\text{CO})) \Rightarrow s \in m(\Sigma_{gp})$$

This condition (together with def.3.1) guarantees that the new formal parameter covers all external signature needs of SPEC, allowing it to replace all previously imported and parameter specifications. A weaker form of generalization may be carried out however, even if this condition is not satisfied, as shown later in this section.

Example 3.3 *Consider again the specification* NATTREE *of example 3.1. The morphism m of example 3.2 trivially satisfies the generalization condition for* NATTREE, *as nat is the only imported sort in* NATTREE *and it is attained by m.*

Definition 3.3 (generalization signature translation) *Given a specification* CO *as described above, a signature monomorphism*[6] *$m : \Sigma_{gp} \to \Sigma$ that satisfies the generalization condition (def. 3.2) for the specification* CO *and its inverse morphism $m^{-1} : m(\Sigma_{gp}) \to \Sigma_{gp}$ define a* generalization signature translation function

[6] injective morphism

$m_g = (f_g : S' \rightharpoonup S'', g_g : Op' \rightharpoonup Op'')$ *and a* generalized signature $\Sigma'' = (S'', Op'')$ *as shown below:*

$$f_g = \begin{cases} \textit{if } s \in m(S_{gp}) \subseteq S \textit{ then } m^{-1}(s) \\ \textit{elsif } s \in S' - S \textit{ then } s \\ \textit{else undefined} \end{cases}$$

$$S'' = S_{gp} \cup (S' - S)$$

$$g_g = \begin{cases} \textit{if } op : s_1..s_n \to s \in m(Op_{gp}) \textit{ then } m^{-1}(op) : m^{-1}(s_1)..m^{-1}(s_n) \to m^{-1}(s) \\ \textit{elsif } op : s_1..s_n \to s \in Op' - Op \textit{ then } op : f_g(s_1)..f_g(s_n) \to f_g(s) \\ \textit{else undefined} \end{cases}$$

$$Op'' = Op_{gp} \cup g_g(Op' - Op)$$

The undefined cases above correspond to sorts and operators of the imported signature Σ that are not attained by the generalization morphism and will disappear in the generalization process if the generalization condition defined above is satisfied.

Example 3.4 *Consider the specification* NatTree *and the (mono)morphism m of example 3.2. Then we have:*

$$\begin{array}{lcl}
S & = & \{nat\} \\
S' & = & \{nat, nattree\} \\
S_{gp} & = & \{s\} \\
S'' & = & S_{gp} \cup (S' - S) = \{s, nattree\} \\
\\
f_g & = & \{nat \mapsto s,\ nattree \mapsto nattree\} \\
\\
Op & = & \{zero : nat,\ suc : nat \to nat, \\
 & & max : nat * nat \to nat, \\
 & & add : nat * nat \to nat\} \\
Op' & = & \{zero : nat,\ suc : nat \to nat, \\
 & & max : nat * nat \to nat, \\
 & & add : nat * nat \to nat \\
 & & empty : nattree, \\
 & & bin : nattree * nat * nattree \to nattree, \\
 & & sum : nattree \to nat\} \\
Op_{gp} & = & \{k : s,\ unop : s \to s,\ bop : s * s \to s\} \\
\\
g_g & = & \{(zero : nat \mapsto k : s),\ (suc : nat \to nat \mapsto unop : s \to s), \\
 & & (add : nat * nat \to nat \mapsto bop : s * s \to s),\ (empty : nattree \mapsto empty : nattree), \\
 & & (bin : nattree * nat * nattree \to nattree \mapsto bin : nattree * s * nattree \to nattree), \\
 & & (sum : nattree \to nat \mapsto sum : nattree \to s)\} \\
\\
Op'' & = & Op_{gp} \cup g_g(Op' - Op) = \{k : s,\ unop : s \to s,\ bop : s * s \to s, \\
 & & empty : nattree,\ bin : nattree * s * nattree \to nattree,\ sum : nattree \to s\}
\end{array}$$

Note: the generalization condition guarantees that $m_g = (f_g, g_g)$ is a morphism from the signature $\Sigma'_{used} \subseteq \Sigma'$ into Σ'', where Σ'_{used} corresponds to the locally declared items (SI', OpI', FD') completed with the part of Σ in the range of m ($m(\Sigma_{gp}) \subseteq \Sigma$).

Definition 3.4 (generalization operation) *The generalized specification*

CO" = `generalize` CO `via` m `with` PARAM

is defined by:

1. *If* CO *is a well formed specification component as in the definitions above;*
2. *If* $m : \Sigma_{gp} \to \Sigma$, *is a signature monomorphism;*
3. *If this morphism defines a specification morphism (that we will also call* m*)* $m : sp_{gp} \to sp$, *where* sp_{gp} *and* sp *are respectively the model denotations of the named specification* PARAM *and the resulting specification of all imports and formal parameter specifications of* CO*;*
4. *If* m *satisfies the* dangling operators condition *(def. 3.1) for* CO*;*
5. *And if* m *satisfies the* generalization condition *(def. 3.2) for* CO*;*

Then, CO" *will correspond to:*

spec CO" [PARAM] = m_g(`SPEC`) ***end***

where m_g(`SPEC`) *is defined as* $(m_g(SI'), m_g(OpI'), m_g(VI'), m_g(FD'), m_g^{\#}(AxI'))$ *such that* $m_g(SI') = f_g(SI')$ *and* $m_g(OpI') = g_g(OpI')$. $m_g(VI')$ *works as if the variables in* VI' *were local constants, adapting only their sorts, if needed;* $m_g^{\#}$ *is the extension of* m_g *to axioms (e.g. in [11]); and* $m_g(FD')$ *is the application of the corresponding translation functions to each component (sort, operator, variable or axiom) of the free-datatype declaration.*

In other words, the generalization operation receives three inputs: the original specification component CO, the potential parameter PARAM and a mapping (signature morphism) from objects in PARAM into the imported objects of CO. This mapping must be injective, so that it can be inversed, and it must correspond to a specification morphism from PARAM into 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, the conditions in definitions 3.1 and 3.2 must hold so 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 *imports* and *params* clauses 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.

Alternatively, condition 5 in definition 3.4 may be omitted. In this case, we get a weaker form of generalization where the original imports and parameter specifications are kept in Co". PARAM is then added as an extra parameter on top of the already existing ones. This implies changes in the translation functions and signatures of definition 3.3 so that f_g and g_g are now total (defined for the whole Σ'); completed with the identity function for the objects that were not previously in their domains. Then, Co" will correspond to:

spec Co" [PARAM] [*params*] **given** *imports* $= m_g$(SPEC) **end**

Example 3.5 *Given specifications* NATTREE *and* PARAM1 *of example 3.1 and the (mono)morphism*

$m : [\textbf{\textit{sorts}}\ s\ \textbf{\textit{ops}}\ k : s, bop : s{*}s \to s] \to [\textbf{\textit{sorts}}\ nat\ \textbf{\textit{ops}}\ zero : nat, add : nat{*}nat \to nat]$

the following generalization command

BTREE = *generalize* NATTREE *via* m *with* PARAM1

generates the specification BTREE *[*PARAM1*] of example 3.1. It would also be possible to generalize*

BTREE2 = *generalize* NATTREE *via* $m2$ *with* PARAM2

where

spec PARAM2 = PARAM1 ***then***
ops $bop2 : s * s \to s$
vars $x, y : s$
- $bop2(x, y) = bop2(y, x)$

end
and

$m2 : [\textbf{\textit{sorts}}\ s\ \textbf{\textit{ops}}\ k : s, bop2 : s * s \to s, bop : s * s \to s] \to$
$[\textbf{\textit{sorts}}\ nat\ \textbf{\textit{ops}}\ zero : nat, max : nat * nat \to nat, add : nat * nat \to nat]$

obtaining

spec BTREE *[*PARAM2*]* =
free type $nattree ::= empty \mid bin(nattree; s; nattree)$
op $sum : nattree \to s$
vars
$a1, a2 : nattree;$
$tn : s;$
- $sum(empty) = k;$ *%(sum.empty)%*
- $sum(bin(a1, tn, a2)) = bop(tn, bop(sum(a1), sum(a2)))$ *%(sum.bin)%*

In this example, we could not have generalized NatTree with a simpler parameter specification such as

spec Elem = **sort** s **end**
via the morphism $m1$ of example 3.2, because the dangling operators condition would not be satisfied (*zero* and *add* would still range over *nat*) and the axioms would not be well formed. E.g., axiom %(d0)% would become

$$sum(empty) = zero$$

equating a term of sort s ($sum(empty)$) with one of sort *nat* ($zero$).

4 Generalizing Components

The generalization operator defined above (section 3) is basically a tool for, once the desired generalization is defined, safely executing the corresponding transformation of the component. The most delicate part in the process is however the definition of the desired level of generality. If we take it to the extreme, any component could be generalized up to the empty specification. Obviously, this kind of result is of no interest, so, we need to fix more reasonable limits. The adopted convention, used in the definition of the generalization operator seen in section 3, is that we can only abstract from imported objects, defining a dual operation to instantiation.

But we still have the choice of how much of the imported specification we want to generalize. First, we can determine which sorts and operations should be generalized, defining a signature. Once this signature is defined, we can decide on the semantic properties of the original specification that we want to keep. If no requirements are made to preserve any semantic properties, the corresponding generalization is called a *syntactic generalization*, otherwise, we call it a *semantic generalization*.

To provide some support on the process of defining the generalization operator parameters, we proposed in [16] a syntactic generalization algorithm and techniques for semantic generalization. In Sections 4.1 and 4.2 respectively we briefly present them.

4.1 Syntactic Generalization

The syntactic generalization is uniquely defined (modulo isomorphism) by the generalized signature. Let Psynt be the formal parameter corresponding to the syntactic generalization of a set of sorts GS in a component Co. Psynt has the structure

spec Psynt =
  sorts $s_1, .., s_n$
  ops $op_1, .., op_m$
  end

I.e., it introduces formal sorts and operators with loose semantics and no equations to be satisfied (the specified class of models is $Alg(\Sigma_{\text{PSYNT}})$).

We reproduce below the algorithm proposed in [16] for, given a set of imported sorts that we want to generalize in a component, define the morphism that corresponds to the maximum generalization that we can obtain. We also showed that this is always possible, given that the set of syntactic generalizations constitutes a complete lattice.

Obtaining the Syntactic Generalization Morphism

Given a set $GS = \{s_1, .., s_n\}$ of sorts (of the resulting imported signature Σ) to generalize in a component CO, the syntactic generalization morphism (and the generalization formal parameter) can be found by the following algorithm:

1. Obtain $GOp = \{op_1, .., op_m\}$, the set of imported operators that occur in any axiom of CO and which have one or more occurences of the generalized sorts in their profiles. These operators will be generalized together with GS.
2. Determine the set $S_{id} = \{s_{n+1}, .., s_{n+k}\}$ of all non-generalized imported sorts which appear in the profiles of these generalized operators.
3. If $S_{id} = \phi$, the generalization is defined for the morphism

   $$m_{\text{SYNT}} : \text{PSYNT}[\textbf{sorts } s''_1, .., s''_n \textbf{ ops } op''_1, .., op''_m] \rightarrow sp[\textbf{sorts } s_1, .., s_n \textbf{ ops } op_1, .., op_m]$$

   that associates the new parameter PSYNT below to sp, the imported specification of CO:

   spec PSYNT =
   sorts $s''_1, ..., s''_n$
   ops $op''_1, ..., op''_m$
   end
4. If $S_{id} \neq \phi$, add it to the set of sorts to generalize and re-execute the algorithm until this set of dangling references (S_{id}) is empty and the generalization, defined.

Example 4.1 *Consider* CO *to be the specification* NATTREE *of example 3.1, and* GS *to be* $\{nat\}$. *Then, we get*

$$GOp = \{zero : nat, add : nat * nat \rightarrow nat\}$$

and $S_{id} = \phi$. *The generalization is then defined for the specification* PARAM1 *of example 3.1 and the morphism*

$$m : [\textbf{\textit{sorts}}\ s\ \textbf{\textit{ops}}\ k : s, bop : s * s \rightarrow s] \rightarrow [\textbf{\textit{sorts}}\ nat\ \textbf{\textit{ops}}\ zero : nat, add : nat * nat \rightarrow nat]$$

Syntactic generalization is often too strong because we loose all semantic properties that depend on properties of the generalized sorts. However, it presents two interesting characteristics: first, it serves as a starting point for semantic generalization (section 4.2); and second, it is this kind of genericity that can be found in programming languages like Ada [2].

4.2 Semantic Generalization

The problem of generalizing a component with preservation of semantic properties represented by equational theorems can be stated as:

> *to generalize a set of imported sorts of a component* Co*, preserving the validity of a given set* Teo *of theorems, in such a way that the original component can be obtained from the generalized one by re-instantiation.*

In [16] there is a broad discussion on this subject. Here, however, we concentrate in a more restricted problem that can be stated as:

> *to generalize a set of imported sorts of a component* Co*, preserving the validity of a given set* P *of proofs of a set* Teo *of theorems, in such a way that the original component can be obtained from the generalized one by re-instantiation.*

This second condition (proof preservation) is obviously stronger than required by the original problem: the validity of a proof implies the validity of a theorem, but the converse is not always true. With this technique, we obtain sufficient but not necessary conditions to the validity of the stated theorems. The main advantage of this technique is its simplicity of implementation, as we substitute the problem of finding a proof for a theorem to the one of validating a known proof.

Independently of the considered problem, including a given set of axioms in the required formal parameter specification of a component makes them valid in all of its models (guaranteed by the semantics of CASL). These axioms can then be used to prove the required theorems. In section 5 we propose an algorithm to generate this set of axioms in the case of rewrite proofs.

5 Preserving Proofs

As stated above, we want to find a property such that: it can syntactically substitute a given imported specification in a specification component Co; and its requirement as formal parameter in the generalized component Co" is enough to guarantee that a given set of proofs for some selected semantic properties of Co is still valid for Co".

These proofs can be of different kinds (rewrite, structural induction, by cases, or others). We concentrate here on rewrite proofs. Because rewrite proofs are intrinsically equational, they are the easiest to preserve. As shown below, it is enough to add a set of equations to the formal parameter so that the proof is still valid after generalization. With other kinds of proofs, as for instance proofs by induction, non-equational (e.g., constructor based) properties of the generalized sorts may be required for the validity of the proof and more sophisticated mechanisms are needed to generalize them.

5.1 Rewrite Proofs

Rewrite techniques [9] are often used in two directions: (1) to prove equational theorems in equational theories and (2) to provide an operational semantics for algebraic specifications of abstract data types (e.g. [5]). In this section, we first introduce our notations for equational theories, rewriting and some particular proofs called *rewrite proofs*. We recall some basic properties used when considering rewrite proofs preservation, and then we show how rewrite proofs can be preserved in the generalized specification. Finally, an algorithm is given to construct a formal parameter and the corresponding morphism so that these proofs are preserved.

5.1.1 Equational Reasoning and Rewriting

Our notations are compatible with the usual ones [9]. Given a first-order signature Σ and a denumerable set of variables $\mathcal{X}$, the set of Σ-terms, denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ is the smallest set containing $\mathcal{X}$ such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\Sigma, \mathcal{X})$ whenever f is an n-ary operator in Σ, and t_i is in $\mathcal{T}(\Sigma, \mathcal{X})$ for $i = 1, \ldots, n$. The terms $t_{|\omega}$, and $t[\omega \leftarrow u]$ denote respectively the subterm of t at the position ω, and the term obtained from t by replacing the subterm $t_{|\omega}$ by u. A Σ-substitution σ is an endomorphism of $\mathcal{T}(\Sigma, \mathcal{X})$ denoted by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where there are only finitely many variables $x_1, \ldots, x_n$ not mapped to themselves. The image of a term t by a substitution σ is denoted by $\sigma(t)$. Given a set E of equational axioms (i.e. pairs of terms denoted by $l = r$), the *equational theory* $=_E$ is the congruence closure of E under the law of substitutivity. Despite of a slight abuse of terminology, E will be often called an equational theory. According to Birkhoff's theorem, $s =_E t$ iff $s = t$ is valid in all models of the class $Alg(\Sigma, E)$. Let $\mathcal{R}$ be the set of *rewrite rules* $l \to r$ such that $l = r \in E$. We write $t \longrightarrow_{\mathcal{R}} t'$ if there exist a rule $l \to r \in \mathcal{R}$, a position ω of t, a substitution σ such that $\sigma(l) = t_{|\omega}$ and $t' = t[\omega \leftarrow \sigma(r)]$. A term t is in *$\mathcal{R}$-normal form* if there is no t' such that $t \longrightarrow_{\mathcal{R}} t'$. We write $t \longleftarrow_{\mathcal{R}} t'$ if $t' \longrightarrow_{\mathcal{R}} t$. It is well-known that $s =_E t$ iff $s \overset{*}{\longleftrightarrow}_{\mathcal{R}} t$, where $\overset{*}{\longleftrightarrow}_{\mathcal{R}}$ is the reflexive-transitive closure of $\longrightarrow_{\mathcal{R}} \cup \longleftarrow_{\mathcal{R}}$. An *equational derivation* (resp. *rewrite derivation*) is any sequence

$$s_0 \longleftrightarrow_{\mathcal{R}} s_1 \longleftrightarrow_{\mathcal{R}} \cdots \longleftrightarrow_{\mathcal{R}} s_n \longleftrightarrow_{\mathcal{R}} \cdots$$

(resp. $s_0 \longrightarrow_{\mathcal{R}} s_1 \longrightarrow_{\mathcal{R}} \ldots \longrightarrow_{\mathcal{R}} s_n \longrightarrow_{\mathcal{R}} \ldots$). An *equational proof* is a finite equational derivation, where each step is "decorated" by the relevant information (applied rewrite rule, and possibly position). A *rewrite proof* of $s = t$ is an equational proof

$$s \stackrel{*}{\longrightarrow}_{\mathcal{R}} v \stackrel{*}{\longleftarrow}_{\mathcal{R}} t$$

such that $s \stackrel{*}{\longrightarrow}_{\mathcal{R}} v$ and $t \stackrel{*}{\longrightarrow}_{\mathcal{R}} v$ are rewrite derivations. It is clear that the existence of a rewrite proof of $s = t$ implies $s =_E t$. Unfortunately, the converse is not always true. But if we assume that $\mathcal{R}$ is terminating (existence of $\mathcal{R}$-normal forms) and confluent (unicity of $\mathcal{R}$-normal forms), then a rewrite proof of $s = t$ exists if $s =_E t$, and rewriting wrt. $\mathcal{R}$ provides a decision algorithm for the equational theory E, due to the following equivalence:

$$s =_E t \Longleftrightarrow s \stackrel{*}{\longrightarrow}_{\mathcal{R}} v \stackrel{*}{\longleftarrow}_{\mathcal{R}} t$$

where v denotes here the unique $\mathcal{R}$-normal form of s and t.

Algebraic specification languages sometimes assume termination and confluence properties of the set of (oriented) equations, so that execution or proofs are feasible in the system. In our context, we assume the existence of a rewrite proof of $s = t$ when $s =_E t$. The equational theory E is derived from the local axioms of a component Co and the axioms of any imported specifications and formal parameters of Co, possibly enriched with already proved theorems of any of these specifications.

5.1.2 Generalizing Rewrite Proofs

For a rewrite proof p of a theorem $s = t$ in a system $\mathcal{R}$ to be valid in a system $\mathcal{R}''$, all rules that are used in p must be present in $\mathcal{R}''$. In our context, $\mathcal{R}$ and $\mathcal{R}''$ are derived from the axioms and already proved theorems of the specifications sp' (corresponding to the original component Co) and sp'' (corresponding to the generalized component Co"). For that reason, if $\mathcal{P}$ is the set of proofs we want to preserve, equations corresponding to all rules that are used in all $p \in \mathcal{P}$ need to be present in the new specification. If they are local equations, this is always true, because the local presentation of the component is not changed (except for possible renamings) by the generalization operator of Section 3. However, if any equations corresponding to imported specifications of Co are used, they may be lost in the generalization process. They must then be added to the formal parameter so that they will be valid in all models of the new specification.

So, let **spec** PSYNT = **sorts** SI_{PSYNT} **ops** OpI_{PSYNT} **end** be the property corresponding to the syntactic generalization of a set of sorts GS in a component Co. Also, let $Eq(p)$ be the set of all equations used in a proof p to be preserved in this generalization. $Eq(p)$ may be split into two sub-sets $Eq^g(p)$ and $Eq^{ng}(p)$, corresponding to generalized (lost) equations and non-generalized equations, respectively. Adding the equations in $Eq^g(p)$ to the property PSYNT, will allow us to reproduce p in the generalized specification. If these equations contain sorts and operators that were not in the signature Σ_{Psynt}, this signature must be expanded.

5.1.3 Obtaining the Generalization Morphism

In the following, we consider the conventions on specifications, signatures and names of section 3, page 9.

Given a set $GS = \{s_1, .., s_n\}$ of sorts (of the resulting imported signature Σ) to generalize in a component Co with preservation of a set of proofs $\mathcal{P}$, the generalization morphism (and the generalization formal parameter) can be found by the following algorithm:

1. Obtain Psynt, the syntactic generalization parameter of GS in Co and the corresponding specification morphism

$$m_{\texttt{SYNT}} : \text{Psynt}[\textbf{sorts } s''_1, .., s''_m \textbf{ ops } op''_1, .., op''_l] \rightarrow sp[\textbf{sorts } s_1, .., s_m \textbf{ ops } op_1, .., op_l]$$

   by the syntactic generalization algorithm (see Section 4.1).

2. Let $GS_{\texttt{SYNT}} = \{s_1, .., s_m\} \supseteq GS$ be the set of generalized sorts after the definition of the syntactic generalization. And let $GO_{\texttt{SYNT}} = \{op_1, .., op_l\}$ be the set of generalized operators.

3. List the set $Eq(\mathcal{P})$ of equations used in $\mathcal{P}$.

4. Identify the subset $Eq^g(\mathcal{P}, GS_{\texttt{SYNT}})$ of equations from $Eq(\mathcal{P})$ that are lost by generalization (external equations that refer to sorts in $GS_{\texttt{SYNT}}$).

$$Eq^g(\mathcal{P}, GS_{\texttt{SYNT}}) = \{e \in Eq(\mathcal{P}) - AxI' | sorts\_of(e) \cap GS_{\texttt{SYNT}} \neq \phi\} = \{e_1, .., e_k\}$$

5. Identify any dangling references $Pend(Eq^g(\mathcal{P}, GS_{\texttt{SYNT}}))$ in $Eq^g(\mathcal{P}, GS)$.

$$Pend(Eq^g(\mathcal{P}, GS_{\texttt{SYNT}})) = sorts\_of(Eq^g(\mathcal{P}, GS_{\texttt{SYNT}})) - GS_{\texttt{SYNT}}$$

6. If $Pend(Eq^g(\mathcal{P}, GS_{\texttt{SYNT}})) = \phi$, then generalization is defined for the morphism

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

   that associates the new parameter Psem below to sp, the imported specification of Co:

spec Psem =
  sorts $s''_1, ..., s''_m$
  ops $op''_1, ..., op''_j$
  • $e''_1, ..., e''_k$
  end

notes:

- $\{op_1, ..., op_j\} = GO_{\mathtt{SYNT}} \cup operators\_of(Eq^g(\mathcal{P}, GS_{\mathtt{SYNT}}))$ and Σ_{PSEM} is well formed because $sorts\_of(operators\_of(Eq^g(\mathcal{P}, GS_{\mathtt{SYNT}}))) = sorts\_of(Eq^g(\mathcal{P}, GS_{\mathtt{SYNT}}))$ and $Pend(Eq^g(\mathcal{P}, GS_{\mathtt{SYNT}})) = \phi$.
- The objects specified by PSEM are the generalized sorts and operators with new names, and the equations are the ones in $Eq^g(\mathcal{P}, GS_{\mathtt{SYNT}})$ with the corresponding remanings.

7. If $Pend(Eq^g(\mathcal{P}, GS_{\mathtt{SYNT}})) \neq \phi$, add it to the set of sorts to generalize and re-execute the algorithm until this set of dangling references is empty and the generalization, defined.

Example 5.1 *Let us consider again the specification of binary trees of natural numbers* NATTREE *of example 3.1 with a given theorem:*

spec NATTREEPROP = NATTREE ***then*** *%implies*
vars *tn: nat ;*
. $sum(bin(empty, tn, empty)) = tn$ *%(th0)%*

*We want to generalize from the sort nat, to specify generic binary trees, but preserving theorem th*0*. Assume then that a rewrite engine (like* ELAN*, see Section 6) has computed a proof p of th*0*:*

$$\begin{array}{ll} sum(bin(empty, tn, empty)) & \rightarrow_{sum.bin} \\ add(tn, add(sum(empty), sum(empty))) & \rightarrow_{sum.empty} \\ add(tn, add(zero, sum(empty))) & \rightarrow_{sum.empty} \\ add(tn, add(zero, zero)) & \rightarrow_{add.0} \\ add(tn, zero) & \rightarrow_{add.0} \\ tn & \end{array}$$

Hence, $Eq(\{p\}) = \{sum.bin, sum.empty, add.0\}$*. The syntactic generalization of nat in* NATTREE *defines the signature* $\Sigma_{\mathrm{PSYNT}} = (\{s\}, \{k : s, bop : s * s \rightarrow s\})$*, corresponding to the sort nat and operators zero and add respectively , which means that* $GS_{\mathtt{SYNT}} = \{nat\}$ *and* $GO_{\mathtt{SYNT}} = \{zero, add\}$*. The equation add.*0 *is the unique element of* $Eq(\{p\})$ *such that (1) it is not locally defined inside* NATTREE *and (2) it is built over an operator involving the sort nat. Hence,* $Eq^g(\{p\}, GS_{\mathtt{SYNT}})$ *is the singleton* $\{add.0\}$*. Since add.*0 *contains only operators of sort nat (namely, zero and add), we have* $Pend(\{add.0\}) = \phi$*, and the generalization is defined for the specification* PSEM *and morphism below.*

spec PSEM =
sorts *s*
ops *k: s ; bop* $: s * s \rightarrow s$ *;*

vars *vars x: s*
- *bop(x,k)* $= x$

end

$$m_{\texttt{SEM}} : \text{PSEM}[\textbf{\textit{sorts}}\ s\ \textbf{\textit{ops}}\ k : s, bop : s * s \to s] \to \text{NAT}[\textbf{\textit{sorts}}\ nat\ \textbf{\textit{ops}}\ zero : nat, add : nat * nat \to nat]$$

It is a property of the proposed algorithm to always produce correct specification morphisms from the new formal parameter into the originally imported specification. In this case, it is easy to verify it: all objects in PSEM *have correspondence in* NAT*; the profiles of operators are trivially compatible with the sort map from s to nat; and the translation of the equation* $bop(x, k) = x$ *of* PSEM *is the equation* $add(x, 0) = x$ *%(add.0)% of* NAT*; and so, all models of the* NAT *specification satisfy it, and their reduct are clearly models of* PSEM.
We can then apply `BTREESEM = generalize NATTREE via` $m_{\texttt{SEM}}$ `with PSEM` *obtaining*

spec BTREESEM *[*PSEM*]* =
free type *btree* ::= *empty* | *bin*(*btree*; *s*; *btree*)
op *sum* : *btree* $\to$ *s*
vars
a1, *a2* : *btree*;
tn : *s*;
- *sum*(*empty*) $= k$; *%(sum.empty)%*
- *sum*(*bin*(*a1*, *tn*, *a2*)) $= bop(tn, bop(sum(a1), sum(a2)))$ *%(sum.bin)%*

which satisfies the generalized theorem $sum(bin(empty, tn, empty)) = tn$.

This specification may now either be re-instantiated into the original NAT *through the same morphism* $m_{\texttt{SEM}}$ *or into another specification such as binary trees of booleans with an operation to compute the and of all node values of a given tree. In this second case, assuming that there is a specification* BOOL *with the usual definitions, it is enough to instantiate* BTREESEM *with the morphism*

$$m_{\texttt{SEM2}} : \text{PSEM}[\textbf{\textit{sorts}}\ s\ \textbf{\textit{ops}}\ k : s, bop : s * s \to s] \to \text{BOOL}[\textbf{\textit{sorts}}\ bool\ \textbf{\textit{ops}}\ T : bool, and : bool * bool \to bool]$$

6 Implementing the Generalization of Rewrite Proofs

In this section, we present the different tools we are using to obtain the components handled by the generalization process. Hence, we use the CASL Tool Set to parse CASL specifications and the ELAN system to normalize CASL expressions. The latter is also used to compute rewrite proofs of theorems that are processed by the semantic generalization algorithm defined in Section 5.1.3.

6.1 The CASL Tool Set

Among the tools being developed for CASL, the main one is the CASL Tool Set (CATS [18]) that consists of a parser, a static checker, a converter to LaTeX. CATS generates different forms of Abstract Syntax Trees (AST):

– **Casfix** The CasFix AST is the result of the parser.

– **CasEnv** The CasEnv AST is produced by the static checker that follows the parsing phase. The FERUS tool makes use of CasEnv. There is a flattened version of CasEnv which is particularly well-suited for the connection of existing tools like theorem provers and rewrite engines, since these tools do not have to consider the different structuring constructs available in CASL. Initially, this flat version of CasEnv was used for ELAN, but we are using also more structured CasEnv since we want to consider CASL specification where theorems are declared as consequences of axioms. In CASL, this kind of specifications is written with a structuring construct.

The content of these Abstract Syntax Trees are then analysed using the ATerm library [8] which is of greatest interest to extract information encoded in term-like data-structures.

6.2 The ELAN system

The ELAN system [7] provides an environment for specifying and prototyping rule-based programs in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms, as it is backed up by the concepts of rewriting and rewriting logic.

The ELAN system is supported by some execution tools – rewrite engines – which are:

- An all-in-one interpreter, written in C++, integrating a pre-processsor step, a parsing step, and an evaluation loop.
- An efficient compiler [17] written in JAVA.
- A new interpreter written in C using the ATerm library.

In a rewrite-based language like ELAN, *executing* a rewrite program means *normalizing* a given query with respect to this rewrite program. Thus, two inputs have to be provided by the user for executing a program:

- A rewrite program including a signature, a set of rules and a set of strategies. In the following, we simply consider the sublanguage of ELAN where the set of strategies is empty.

- A *query* term expressed in the signature of the rewrite program.

Given these inputs, ELAN computes the normal forms of the query term, with respect to the rewrite program. Note that because the set of rules is not required to be terminating nor confluent, a query term may have several normal forms, or may not terminate.

Actually, we may find three different kinds of syntax in the ELAN system for rewrite programs:

- A user-friendly syntax that allows us to use mixfix notation for terms in the program. For this syntax, we need a sophisticated earley-based mixfix parser integrated in the interpreter.
- A "computer-friendly" syntax, to represent the internal structure of a rewrite program. This syntax, called REF, is rather difficult to handle directly since it looks like an encoding of the program [6]. It provides a textual representation of the ELAN working memory associated to a rewrite program. In a REF program, we may find some tables for identifiers, module names, sort names, rule names and strategy names, some grammar rules for the mixfix parser, the encoding of rules, strategies, and the query term. A REF program is usually generated by the ELAN interpreter, and can be executed both by the interpreter and the compiler.
- An abstract syntax, called Efix, which is generated by a new ELAN parser. This abstract syntax aims at being used as an exchange format for tools developed in the ELAN environment, but also in connection with tools of other systems, as shown in this paper. The new ELAN interpreter implemented using ATERMs is based on this abstract syntax: it normalizes an input query (Efix) term with respect to an Efix rewrite program.

6.3 A Rewrite-based Tool Support for the Generalization

We briefly describe how to integrate CATS and ELAN into the FERUS environment, and what has to be done for using ELAN rewrite engines as a support for the generalization.

6.3.1 Executing with ELAN

To connect CATS and ELAN, a translation tool has been implemented for transforming a CASL AST of the form CasEnv into the ELAN AST (Efix). This tool called `env2efix` is implemented in `C` with the ATERM library. It supports CASL mixfix operators and binary CASL operators declared as Associative-Commutative. Then, we use the ELAN parsing technology to parse queries in mixfix notation, and the

ELAN rewrite engines to perform rewriting modulo the Associativity-Commutativity equational theory.

Using this translation tool on the outputs of CATS, we are able to call rewrite engines initially developed for ELAN, which can be called uniformly via a command `casl2elan` running successively CATS, a translation tool, and then the rewrite engine chosen by the user:

- the ELAN interpreter evaluates queries with respect to a REF rewrite program obtained after the sequential call of CATS;`env2efix`;`efix2ref`. A new ELAN interpreter based on ATerms is being developed. It behaves like the "old" ELAN interpreter, but it simply requires an input Efix rewrite program, obtained by the sequence CATS;`env2efix`.
- the ELAN compiler generates an executable program from a REF rewrite program obtained after applying sequentially CATS;`env2efix`;`efix2ref`.

All these rewrite engines aim at computing the normal form of a given query expression with respect to a rewrite program R (obtained from the translation of a CASL equational specification). The query can be any term: for example, it is possible to consider a query of the form `s=t`, where `=` is the ELAN built-in equality predicate, in order to check if $s = t$ is or not a theorem of R. The normal form of such query is `true` if $s = t$ is a theorem of R and `false` otherwise.

6.3.2 Proving with ELAN

In the FERUS environment, we not only want to use ELAN as a theorem-checker, but we may also be interested in the rewrite proof leading to the result `true` or `false`. For this reason, we need that ELAN returns the result but also, as side effect, the rewrite derivation leading to the result. In our first experiments, we have created a "proof mode" for ELAN by modifying the new ELAN interpreter based on ATerms in such a way that it simply returns the set of rules involved in the rewrite derivation, using the statistics computed by the rewrite engine (for each rule, the number of rule applications). Indeed, for the generalization process, the *set* of applied rules is sufficient, the algorithm does not need to know the *sequence* of applied rules. By abuse of language, this set is called in the following the computed rewrite proof.

In addition to this slight modification of ELAN, we need to extract axioms (rewrite rules of R) and theorems (query expressions) from the output of CATS. This can be done by considering CasEnv AST where the `%implies` annotation indicates that some equational formulas (the theorems) are consequences of another set of equational formulas (the axioms). Axioms are translated into Efix rewrite rules, whilst theorems – the proof goals – are given as input queries after skolemization of variables into free constants. Given the corresponding Efix rewrite program, and for each input query term, ELAN computes a result together with a rewrite proof. This leads to a set of rewrite proofs that can be encoded as a list of ATerms. After the

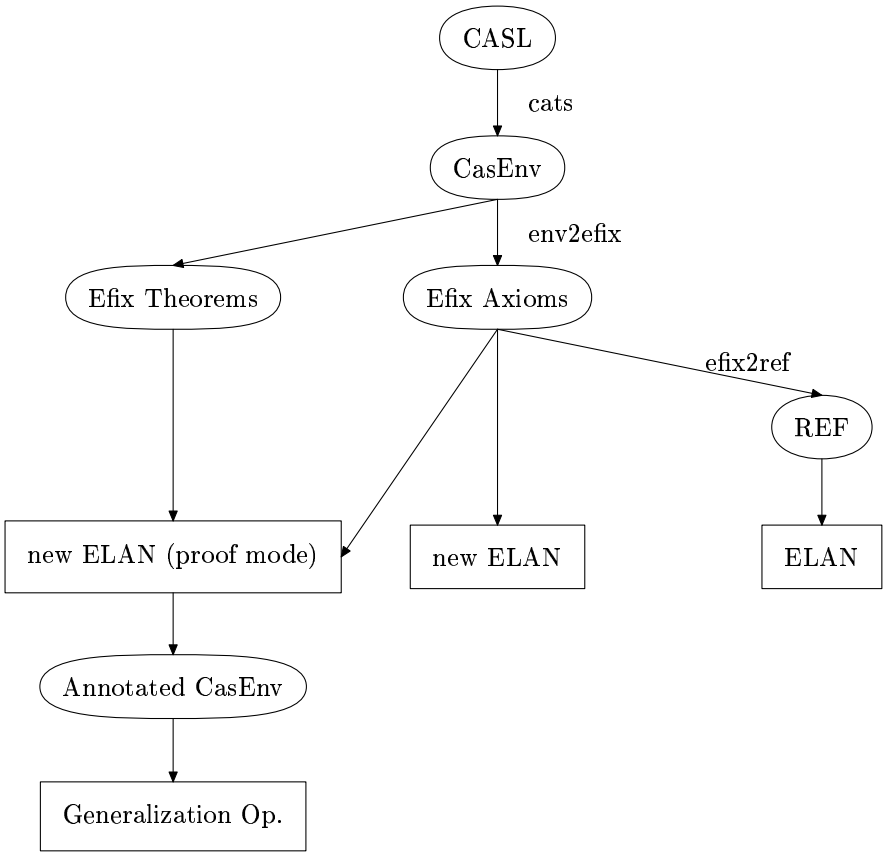

Figure 1: Using ELAN rewrite engines for the generalization

computation by ELAN of rewrite proofs of theorems, we still have to translate back these proofs to CASL. To this end, we are developing a tool to annotate any rewrite-proved equational formula occurring in CasEnv AST with a dedicated annotation, let us say `%proved_by`, that will include the related rewrite proof. The obtained CasEnv AST will be processed by the FERUS tool, and the `%proved_by` annotation will be used by the semantic generalization algorithm as decribed in Section 5. For instance, the theorem $\%(th.0)\%$ of example 5.1, would be annotated by its "proof", containing the list of formulas

$$\%(sum.bin)\%, \%(sum.empty)\%, \%(add.0)\%$$

The general architecture of the ELAN proof mode used as a support of generalization is depicted in Figure 1.

7 Generalizing Theorems

The semantic generalization introduced in Section 4 relies on the problem of constructing a new generalized component from an original component, by preserving the validity of a given theorem (or a given set of theorems). In Section 5, we first concentrated on equational theorems for which we assume the existence of rewrite proofs, possibly computed by a rewrite engine as shown in Section 6. In this particular case, it suffices to consider the problem of preserving rewrite proofs. In the general case, we can be interested in preserving a theorem $s = t$ for which there is no rewrite proof. In this section, our aim is not to give a general solution to this problem. We just want to sketch a simple method based on the application of rewriting techniques together with the notion of *variable abstraction*, a well-known combination technique for purifying an equation involving operators from different theories. Here, we want to map the original equation $s = t$ into a valid equation involving only generalized (lost) operators.

To define precisely the notion of *variable abstraction*, let us first introduce some notations and objects:

- The signature of Co can be split into two subsignatures Σ_g and Σ_{ng}, corresponding respectively to the generalized (lost) signature and to the non-generalized signature.
- $Aliens(s = t)$ is the set of subterms of s and t headed by operators in Σ_{ng} such that all their superterms are headed by operators in Σ_g.
- $\equiv$ is an equivalence relation on $Aliens(s = t)$, such that $u \equiv u'$ implies $u = u'$ is valid in the component Co. The $\equiv$-equivalence class of a term u is denoted by $[u]_{\equiv}$.

- π is a bijective mapping from $Aliens(s = t)/_{\equiv}$ to $\mathcal{AV}$, a set of new variables different from variables in s and t.

Definition 7.1 *The* variable abstraction *of s (resp. t) is denoted by s^π (resp. t^π) and defined inductively as follows.*

- $(f(t_1, \ldots, t_n))^\pi = f(t_1^\pi, \ldots, t_n^\pi)$ *if* $f \in \Sigma_g$,
- $(f(t_1, \ldots, t_n))^\pi = \pi([f(t_1, \ldots, t_n)]_\equiv)$ *if* $f \in \Sigma_{ng}$,
- $x^\pi = x$ *if x is a variable.*

Example 7.1 *Consider the equation*

$$add(X, add(Y, sum(L))) = add(Y, add(X, sum(L)))$$

Assume that add is in Σ_g and sum is in Σ_{ng}. The variable abstractions of the left-hand side and the right-hand side are respectively

$$add(X, add(Y, Z)) \quad and \quad add(Y, add(X, Z))$$

where $Alien(e) = \{sum(L)\}$ and $\pi(sum(L)) = Z$.

We will illustrate our method on the following specification:

Example 7.2 ***spec*** NATLIST = NAT ***then***
free type $list ::= nil \mid cons(nat; list)$
op $sum : list \to nat$
vars
$L : list$;
$X : nat$;
- $sum(nil) = zero$; *%(sum.nil)%*
- $sum(cons(X, L)) = add(X, sum(L))$ *%(sum.cons)%*

Given this specification, we want now to preserve in the generalized component CO" the theorem that states that the order of elements in a sequence has no influence in the resulting sum:

$$sum(cons(X, cons(Y, L))) = sum(cons(Y, cons(X, L)))$$

The method for preserving the validity of $s = t$ in the new component CO" is described below in an informal way.

1. Compute the syntactic generalization of CO for a set of sorts GS. This gives us PSYNT, Σ_g and Σ_{ng}.

2. Simplify $e := (s = t)$ by rewriting s and t into s_g and t_g, such that s_g and t_g are headed by operators in Σ_g. This leads to the equation $e_g := (s_g = t_g)$. For sake of simplicity, we assume that the rewrite-based simplification only involves non-generalized equations.

   Example 7.3 *(Example 7.2 continued) Applying repeatedly rewrite rules defined in* NatList*, permits us to simplify the members of the equation e, and to obtain*

   $$e_g := add(X, add(Y, sum(L))) = add(Y, add(X, sum(L)))$$

   Because the original equation is a theorem in NatList*, the equation e_g obtained by rewriting simplification is still valid in* NatList*, for all possible natural values of X and Y and natural list values of L.*

3. Compute the variable abstraction of s_g and t_g. This leads to the equation $e_g^\pi := (s_g^\pi = t_g^\pi)$.

   Example 7.4 *(Example 7.3 continued) The variable abstraction of $add(X, add(Y, sum(L)))$ and $add(Y, add(X, sum(L)))$ yields the equation*

   $$e_g^\pi := add(X, add(Y, Z)) = add(Y, add(X, Z))$$

4. Provided that e_g^π is valid in the imported specification lost by generalization, add e_g^π to Psynt.

   Example 7.5 *(Example 7.4 continued) We must verify that e_g^π is valid in* Nat*. This verification is needed to guarantee the existence of a specification morphism from the parameter to* Nat*. This condition is necessary to guarantee that the original* NatList *specification is a specialization of the generalized one. In this case, e_g^π is valid in* Nat *because sum is a surjective function in nat:*

   $$\forall Z : nat\ \exists L : \texttt{seq\_nat},\ sum(L) = Z$$

   Then, (a renaming of) e_g^π is added to Psynt*, thus leading to*

   spec Psem2 =
   sorts *s*
   ops *k: s ; bop: s * s -> s ;*
   vars *vars X, Y, Z: s*
   • *bop(X,bop(Y,Z)) = bop(Y,bop(X,Z))*
   end

With this method, we obtain a parameter Psem2 such that the theorem $s = t$ is still valid in the component Co" obtained by generalizing NatList with Psem2.

8 Future Works and Conclusions

This paper presents part of our work concerning the generalization by parameterization of algebraic specification components for reuse. The main result presented here is the proposal of an algorithm to identify the formal parameters that guarantee the validity of some semantic properties through their rewrite proofs. This algorithm is to be applied in conjunction with the generalization operator (also defined in this paper) that safely performs the generalization transformations in the component. This combination provides the means to obtain a more general specification component from which the original one is a specialization and that still satisfies a given set of equational properties with their rewrite proofs.

It was not shown here, but more complex proofs can benefit from this result, although only partially. One of the next steps in this work is to improve the treatment of these more complex kinds of proofs that may need sort-generation properties to be required on the formal parameter. One possible direction to take is the use of subsorts and a larger sublanguage of CASL, so that some sort generation based properties can be preserved in the generalization of a sort into a super sort.

On a different direction, we are developing a prototype for a specification component manipulation tool using these ideas in the context of project FERUS and will produce a case study. This case study should be the starting point for the proposal of a methodology of support to the creation of component libraries, which is the practical main goal of this work.

References

[1] J.-R. Abrial. *The B-Book — Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Alsys. *Reference Manual for the Ada programming language*, 1983.

[3] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[4] S. Autexier and T. Mossakowski. Integrating HOL-CASL into the Development Graph Manager MAYA. In Alessandro Armando, editor, *Proc. FroCoS'2002*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 2–17. Springer-Verlag, April 2002.

[5] D. Bert and R. Echahed. On the operational semantics of the algebraic and logic language LPG. In *Recent Trends in Data Types Specification*, number 906 in Lecture Notes in Computer Science, pages 132–152. Springer-Verlag, 1995. Selected paper from 10th Workshop on Specification of Abstract Data Types.

[6] P. Borovanský, S. Jamoussi, P.-E. Moreau, and Ch. Ringeissen. Handling ELAN Rewrite Programs via an Exchange Format. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), Sep. 1998. Elsevier.

[7] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and Ch. 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, Pont-à-Mousson (France), Sep. 1998. Elsevier.

[8] 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.

[9] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theorectical Computer Science*, chapter 15. Elsevier Science Publishers B.V., 1990.

[10] H. Ehrig, W. Fey, and H. Fansen. ACT ONE: an algebraic specification language with two levels of semantics. Technical Report Technical Report 83-03, Technical University of Berlin, 1983.

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

[12] CoFI group. *CASL, The Common Algebraic Specification Language — Semantics*. Available at the CoFI home page: http://www.briks.dk/Projects/CoFI.

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

[14] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Language and Tools for Formal Specification*. Springer-Verlag, 1993.

[15] H. Kirchner and C. Ringeissen. Executing CASL Equational Specifications with the ELAN Rewrite Engine. Technical report, CoFI Document, 2000. Note T-9.

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

[17] P.E. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, Sep. 1998.

[18] T. Mossakowski. Casl - from semantics to tools. In *Proceedings of TACAS 2000*, number 1785 in LNCS, pages 93–108, Berlin, 2000. Springer Verlag.

[19] 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.

[20] T. Mossakowski, K. Kolyang, and B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In *Workshop on Algebraic Development Techniques (WADT'97), Selected Papers*, number 1376 in Lecture Notes in Computer Science, pages 333–348. Springer Verlag, 1997.

[21] Lutz Schröder and Till Mossakowski. HASCASL: Towards integrated specification and development of functional programs. In *Proceedings of AMAST 2002*, number 2422 in Lecture Notes in Computer Science, pages 99–116, Berlin, 2002. Springer Verlag.

[22] J.M. Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall International Series in Computer Science. Prentice Hall Int., 2nd edition, 1992.

[23] M.G.J. van den Brand and J.S. Scheerder. Development of Parsing Tools for CASL using Generic Language Technology. In D. Bert and C. Choppy, editors, *Workshop on Algebraic Development Techniques (WADT'99), Selected Papers*, volume 1827 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2000.

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

A FERUS Abstract Syntax Grammar

```
------------------------------------------------------------------
FERUS Basic Specifications - Abstract Syntax
------------------------------------------------------------------

BASIC-SPEC       ::= basic-spec BASIC-ITEMS*

BASIC-ITEMS      ::= SIG-ITEMS | FREE-DATATYPE
                   | VAR-ITEMS | AXIOM-ITEMS

------------------------------------------------------------------

SIG-ITEMS        ::= SORT-ITEMS | OP-ITEMS

------------------------------------------------------------------
```

```
SORT-ITEMS       ::= sort-items SORT-ITEM+
SORT-ITEM        ::= SORT-DECL

SORT-DECL        ::= sort-decl SORT+

-----------------------------------------------------------------

OP-ITEMS         ::= op-items OP-ITEM+
OP-ITEM          ::= OP-DECL

OP-DECL          ::= op-decl OP-NAME+ OP-TYPE
OP-TYPE          ::= TOTAL-OP-TYPE
TOTAL-OP-TYPE    ::= total-op-type SORT-LIST SORT
SORT-LIST        ::= sort-list SORT*

-----------------------------------------------------------------

DATATYPE-ITEMS   ::= datatype-items DATATYPE-DECL+
DATATYPE-DECL    ::= datatype-decl SORT ALTERNATIVE+
ALTERNATIVE      ::= TOTAL-CONSTRUCT
TOTAL-CONSTRUCT  ::= total-construct   OP-NAME COMPONENTS*
COMPONENTS       ::= SORT

-----------------------------------------------------------------

FREE-DATATYPE    ::= free-datatype DATATYPE-ITEMS

-----------------------------------------------------------------

VAR-ITEMS        ::= var-items VAR-DECL+
VAR-DECL         ::= var-decl VAR+ SORT

-----------------------------------------------------------------

AXIOM-ITEMS      ::= axiom-items AXIOM+

AXIOM            ::= FORMULA
FORMULA          ::= ATOM | IMPLICATION
IMPLICATION      ::= implication ATOM CONJUNCTION

CONJUNCTION      ::= conjunction ATOM+
```

```
                    | ATOM

ATOM              ::= STRONG-EQUATION
STRONG-EQUATION   ::= strong-equation TERM TERM

-----------------------------------------------------------------

TERMS             ::= terms TERM*
TERM              ::= SIMPLE-ID | QUAL-VAR | APPLICATION
                    | SORTED-TERM
QUAL-VAR          ::= qual-var VAR SORT
APPLICATION       ::= application OP-SYMB TERMS
OP-SYMB           ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME      ::= qual-op-name OP-NAME OP-TYPE
SORTED-TERM       ::= sorted-term TERM SORT

-----------------------------------------------------------------

SORT              ::= TOKEN-ID
OP-NAME           ::= ID
VAR               ::= SIMPLE-ID

-----------------------------------------------------------------

SIMPLE-ID         ::= WORDS
ID                ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID          ::= TOKEN
TOKEN             ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
MIXFIX-ID         ::= TOKEN-PLACES
TOKEN-PLACES      ::= token-places TOKEN-OR-PLACE+
TOKEN-OR-PLACE    ::= TOKEN | PLACE

-----------------------------------------------------------------

-----------------------------------------------------------------
FERUS Structured Specifications - Abstract Syntax
-----------------------------------------------------------------

SPEC              ::= BASIC-SPEC | EXTENSION
EXTENSION         ::= extension SPEC+

SPEC-DEFN         ::= spec-defn SPEC-NAME GENERICITY SPEC
```

```
GENERICITY       ::= genericity PARAMS IMPORTED
PARAMS           ::= params SPEC*
IMPORTED         ::= imported SPEC*

SPEC-NAME        ::= SIMPLE-ID
```

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399