Automatic generation of functional programs from CASL specifications

Agnès Arnould, Marc Aiguier, Laurent Fuchs, Thibaud Brunet

Automatic generation of functional programs from CASL specifications

Agnès Arnould*, Laurent Fuchs*, Marc Aiguier† and Thibaud Brunet*
* SIC, Université de Poitiers, SP2MI, 86962 Futuroscope, France, Email: {arnould,fuchs}@sic.univ-poitiers.fr
† LaMI, CNRS-UMR 842, Université d'Évry, 91000 Évry, France, Email: aiguier@lami.univ-poitiers.fr

***Abstract*— In this paper, we present a code generator transforming a class of CASL specifications into O'Caml programs. This code generator is dedicated to rapid prototyping of CASL specifications especially in the area of geometric modeling where algebraic formalisms have been used since the last decade.**

A large class of constructive equational specifications is handled by this generator while insuring the correctness of generated O'Caml programs. In particular, CASL specifications with many interpretation models (i.e. incomplete) are automatically supplemented in order to produce a program that implements one of them. Underlying properties, such as termination, completeness and confluence hold when equations satisfy some syntactic criteria given in the paper.

I. INTRODUCTION

Formal methods are no longer confined to specify critical systems (transport, energy, space). They are also intensively used in various competitive areas such as telephony, electronics trades, micro-electronics. The reason is they are equipped with various tools (proof, test, code generation, etc.) that allow to develop working-order software with an optimal trade-off of cost, deadline and performance. For several years, various authors [1], [2], [3], [4], [5], use specification formalisms to specify systems in geometric modeling (mechanical computer-assisted designing, cartoon movies, surgical simulators). Different specification case studies have been undertaken in geometric modeling using respectively the oriented-model language B [5] and the last-born of algebraic languages CASL (the Common Algebraic Language Specification) [4], [6]. These different case studies have shown a better adequacy of the language CASL to specify systems in this area. This is due to basic mathematical structures which are manipulated in geometric modeling (such as G-maps [7] or simploidal sets [8]). Indeed, these mathematical structures are algebraic structures (i.e. sets together with functions and distinguished elements), and CASL is an algebraic language dedicated to specification of algebraic structures. Moreover, CASL allows to specify more abstractly systems than B but also to refine specifications up to obtaining concrete ones. This need of abstraction becomes essential in geometric modeling where complexity of manipulated data and related algorithms is always increasing to be closer to reality.

CASL is still equipped with few tools (mainly because of the youth of the language). For prototyping CASL specifications, two tools exist. The first tool has been obtained by connecting to CASL the system ELAN that performs rewriting systems associated to "concrete" specifications [9]. The problem with this tool is it cannot be directly used in our geometric modeling applications. In the targeted applications, the generated codes must be interfaced with existing modelers (usually written in C or C++), particularly with their graphic interface. Recently, in [10] a second tool has been developed which allows to generate functional programs written in Haskell from a subclass of CASL specifications. Interest of functional programming languages, such as Haskell or O'Caml, comes from their syntactical proximity to CASL and enable interfacing with other programming languages. This second tool implements a previous work of T. Mossakowski [11] where two sub-languages of CASL have been defined whose specifications satisfying a set of syntactical constraints can be directly transformed into convergent rewriting systems (i.e. confluent and terminating). Specifications written in these two sub-languages are, roughly speaking, complete specifications manipulating defining equations (see Section III). The drawback with the tool developed in [10] is the subclass of CASL specifications considered to generate Haskell programs is larger than the two sub-languages developed in [11]. Generated programs are then no longer insured to be correct with respect to their specifications, and terminating. However, no condition or proof obligation have been imposed or generated to insure such a correctness. In this paper, we then propose to extend this work in order to answer this drawback. Hence, we propose to develop a code generator which transforms a subclass of CASL specifications into O'Caml programs[1]. This will be achieved both by extending the two sub-languages developed in [11] to incomplete specifications and by automatically checking correctness of generated programs. Incomplete specifications are considered because in geometric modeling we are often confronted with such specifications. For instance, the rounding operation[2] is standard and classically used in mechanical design. However, it is naturally incomplete, and then may produce many different rounding surfaces. The choice of one of them is arbitrary and depends on esthetic criteria. Of course, when dealing with incomplete specifications, many models and then implementations are possible. To choose one of them, unspecified cases of total operations will then be supplemented by allocating them to any correct value (see Section IV-B.3).

The paper is structured as follows: in Section II, both syntax

[1]The choice of O'Caml is because applications in geometric modeling developed in the laboratory of the two first authors of this paper are written in O'Caml.

[2]This operation consists on replacing a sharp edge by a curved surface.

and semantic of the useful part of CASL is briefly presented. Section III is devoted to detail the two constructive sub-languages from which O'Caml programs are generated. First, the initial proposition of CASL sub-languages defined by T. Mossakowski [11] is presented. After, to deal with a larger family of specifications, our extension of these languages is detailed. Some hints on fundamentals results (termination, completeness, and confluence) insuring correctness of generated programs are given. Rigorous proofs of these results can be found in [12]. Finally, in Section IV, the implementation of code generator is presented.

II. PRESENTATION OF CASL

In this Section, we briefly present the aspects of CASL language that we use in the paper. For a complete presentation of CASL, interested readers can refer to [13]. All concepts are illustrated by the same didactic example from which O'Caml code will be generated. For lack of space, we cannot give examples of geometric modeling. However, examples of such specifications can be found in [2], [4], [5], [6]

sorts *Basic*, *Color*
ops *Red*, *Blue*, *Green* : *Basic*;
Black : *Color*;
mix : *Basic* × *Color* → *Color*;
_@_ : *Color* × *Color* → *Color*;
basic : *Color* →? *Basic*
pred *basic* : *Color*

Fig. 1. Signature

First, a CASL specification is defined by a *signature* and a set (usually finite) of well-formed formulæ, called *axioms* built on this signature. Signatures consist of:

- a set of sorts which are names of data types introduced by the keyword **sort(s)** . In our example, Fig. 1, we have two sorts, $Basic$ and $Color$ that respectively represent basic colors and general colors,
- a set of operation names introduced by the keyword **op(s)** . In the above example we have 4 constants, Red, $Blue$, $Green$ that are basic colors and $black$ that is a color, 2 total operations mix and _@_ which respectively make possible to mix a basic color and general color with another general color, and a partial operation $basic$, denoted by →?, that returns the basic color of a color made up of only one basic color,
- a set of predicate names introduced by the keyword **pred(s)** . In our example we have a predicate $basic$ which holds when a color is restricted to a basic one.

Both operation and predicate names are equipped with profiles of the form $s_1 \times \ldots \times s_n \rightarrow s_{n+1}$ for operations and $s_1 \times \ldots \times s_n$ for predicates. In CASL, multifixe symbols can be defined, such as the operation _@_, and names can be overloaded as $basic$ which represents both the operation and the predicate.

Mathematical meaning of signature elements are; for each sort, a set of values, for each total (resp. partial) operation, a total (resp. partial) mapping defined on the sets of values by respecting operation profile, and for each predicate, a relation defined on the Cartesian product of value sets by respecting predicate profile.

Thus the signature of Fig. 1 has many possible models such as the following one: $Basic$ is a set of paint tubes containing at least three ones respectively labeled Red, $Blue$ and $Green$; $Color$ is the set of all the colors that one can obtain on the pallet by mixing together tubes of paint; $black$ is the empty pallet before any addition of painting; mix adds an amount of a paint tube on a pallet; _@_ mixes the content of two pallets; and the operation $basic$ removes the pallet content when it is only composed of a basic color. Another model of above signature is a computer screen with liquid crystals: $Basic$ is then the set of three kinds of screen crystals (Red, $Blue$ and $Green$); the set of values associated to the sort $Color$ is obtained by varying the intensity of crystals light (mixtures are recomposed by human eye).

∀ *b*: *Basic*; *c*, *c1*, *c2*: *Color*
• *Black* @ *c* = *c* **%**(@Black)**%**
• *mix*(*b*, *c1*) @ *c2* = *mix*(*b*, *c1* @ *c2*) **%**(@mix)**%**

Fig. 2. Axioms

Axioms are first-order formulæ. They are built on signature elements. They express the required properties of the specified system. For example, in Fig. 2, two axioms built on the signature of Fig. 1 are given, that recursively define _@_ from both operations $Black$ and mix. In order to refer to the axioms, names can be given (respectively @Black and @mix in our example, see Fig. 2).

A model of a CASL specification is then a model of the signature that satisfies all its axioms (according the standard satisfaction relation ⊨ in the first-order logic). In our example, the first model of the painting pallet satisfies all the axioms. So, the empty pallet $Black$ is neutral for mix and the order of application of mix and _@_ to compose pallets is insignificant. Similarly, our second model also satisfies both axioms of Fig. 2. Semantics of the specification is then "loose" since it has, at least, these two models.

Together with axioms, generation constraints can be added for one or many sorts with respect to some operations of the signature. Generation constraints allow to cut down in specification model class. All models that contain values not corresponding to the evaluation of ground terms are eliminated. In CASL, these constraints are introduced by the keyword **generated type**. For example:

generated type *Basic* ::= *Red* | *Blue* | *Green*

we impose that values of the sort $Basic$ are the meaning of the three constants Red, $Blue$ and $Green$. With this supplementary constraint, any model of the pallet is no longer a model of our specification when its number of paint tubes exceeds three. On the opposite, the screen model is a model that satisfies the generation constraint. Indeed, there is only three different types of crystals. Black and white screens are also possible models where the three constants Red, $Blue$ and $Green$ are interpreted by the same kind of grey crystal.

Generation constraints can be reinforced by imposing that sorts are *freely generated* by some operations. In this case, these operations are usually called *generators*. In CASL, this is introduced by the keyword **free type**. For example, writing

free type *Basic* ::= *Red* | *Blue* | *Green*

means that all models of the specification have a value set for the sort $Basic$ with exactly three values, each one being the interpretation of the three constants Red, $Blue$ and $Green$. Hence, the value set in any model is isomorphic to the ground terms of the sort only built from the associated generators.

CASL offers various syntactic facilities and structuring elements. Structured specifications can be normalized to basic specifications described above by using the Hets analysis tool. Here, we do not detail these features as we do not take them directly into account during the code generation, suggested reference for interested reader is [13].

III. CONSTRUCTIVE SUB-LANGUAGE OF CASL

A. State of the art

Here, we present the two sub-languages of CASL as defined in [11]. Both are restricted to free types, operations (predicates are not considered) and equations. Moreover, equations are equipped with the following particular form:

$$f(u_1, ..., u_n) = t$$

where $u_1..., u_n$ are terms only built from generators and their variables have only one occurrence, and t is a term built on the signature such that its variables are among the variables of $u_1..., u_n$. Such equations are called *defining equations*. These equations are oriented from left to right. This gives rise to a rewriting system. Supplementary syntactical constraints are then imposed on defining equations to obtain rewriting systems which are complete (i.e. any term containing non-constructors is reducible), terminating (i.e. there does not exist infinite descending by the closure of the rewriting relation $\xrightarrow{*}$[3]) and confluent[4] (i.e. term reducing by rewriting is deterministic).

The first and the second sub-language defined in [11] only manipulate total and partial operations, respectively. Hence, in the second sub-language, the partiality of operations means that some values may yield nonterminating evaluations, and then only the confluence property is checked on the associated rewriting system. However, specifications must be under the scope of a free extension [13], which is a structuring element of CASL. In our context, this leads to consider that partial operations are undefined for every value which is not specified. Hence, any ground term with an infinite reduction or a normal form which is not only built from generators, will be considered as undefined. The goal of T. Mossakowski in this first work [11] was to propose sub-languages whose specifications have a single computable model (up to isomorphism) that is they are complete.

[3]that is it is a well founded relation.

[4]A binary relation $\rightarrow$ included in $A \times A$ is *confluent* if and only if $\xleftarrow{*} \circ \xrightarrow{*}$ is included in $\xrightarrow{*} \circ \xleftarrow{*}$, where "$\circ$" and $\xrightarrow{*}$ indicate, respectively, the relation composition and the reflexive and transitive closure of $\rightarrow$.

In the following, we extend this first work in order to consider a larger sub-language by including incomplete specifications and predicates.

B. Data structures

As in [11], we consider only free types (i.e. sorts introduced by **free type** - see Section II). Value sets are then isomorphic to ground terms built on generators. Hence, CASL free types exactly correspond to sum types of O'Caml.

free type
Basic ::= *Red* | *Blue* | *Green* ; **%%** Basic colors
Color ::= *Black* | *mix*(*Basic*; *Color*) **%%** Other colors

Fig. 3. Data type

Fig. 3 defines previous sorts $Basic$ and $Color$ (see Fig. 1) as free types. For each set of generators, the (hidden) axioms that determine free generation are imposed.

C. Total operations

In [11], both sub-languages are equipped with strong syntactical constraints on equations. Then, this leads to specifications with a unique model (up to isomorphism). Here, these syntactical constraints are weakened, especially completeness. Consequently, a possible model in the model class of specifications must be chosen. This is done by supplementing the specifications (see Section IV). Moreover, when both termination and confluence fail, proof obligations will be generated and left to specifier responsibility.

1) Termination: For total operations, the underlying O'Caml program must necessarily be terminating to be correct. This property is insured using a standard rewriting technique [14]: the *recursive path ordering* (rpo). This order requires first a well-founded order, called precedence order and denoted by $<$, on signature operations. In Section IV, we will see how to generate this precedence order by building a dependence graph of operations from specification under prototyping. Hence, in addition to syntactical constraints associated to defining equations, a supplementary one is imposed: every defining equation $f(u_1..., u_n) = t$ must satisfy, for the precedence order $>$, that no sub-term t' of t is labeled with a head operation g such that $g > f$. When this supplementary condition is satisfied, we have shown in [12] that the underlying rewriting system obtained by orienting from left to right equations is terminating.

Axioms in Fig. 4 are defining equations that satisfy the supplementary condition from the precedence ordering:

$$\_\_@\_\_ > mix, red > Black, red > Red, red > mix$$

and then, the underlying rewriting system is terminating.

When there does not exist a precedence order, $>$, on signature operations, proof obligations are generated. These proof obligations could be discharged by the user using either the prover associated with CASL or some other specialized tools (Cime).

ops $\_@\_ : Color \times Color \rightarrow Color$; %% Concatenation.
$red : Color \rightarrow Color$ %% Extract all reds of a color.

$\forall$ *b*: *Basic*; *c*, *c1*, *c2*: *Color*
• $c \ @ \ Black = c$ %(Black@)%
• $Black \ @ \ c = c$ %(@Black)%
%% Basic cases
• $mix(b, c1) \ @ \ c2 = mix(b, c1 \ @ \ c2)$ %(@mix)%
%% Recursive call

• $red(Black) = Black$ %(redBlack)%
• $red(mix(Red, c)) = mix(Red, red(c))$ %(redRed)%
• $red(mix(Blue, c)) = red(c)$ %(redBlue)%
• $red(mix(Green, c)) = red(c)$ %(redGreen)%

Fig. 4. Total operations

2) Confluence: When the rewriting system associated to defining equations is not confluent, this means that some ground terms t have several different normal forms. If two normal forms only built from generators exist for a ground term, then the set of defining equations is inconsistent. In this case, a correct program cannot be generated. In another case, when the set of defining equations have some normal forms with some non-generator operations f, then this means that f is insufficiently specified.

As in [11], confluence is insured by imposing that left members of defining equations are not pairwise unifiable(recall that unification is computable) that is the set of critical pairs is empty. So, by Knuth-Bendix's lemma, the underlying rewriting system is locally confluent and then, when it is terminating, it is also confluent by Newman's lemma. For example, axioms in Fig. 4 specify operations red on four distinct sub-domains since left members of equations are not unifiable.

When this criterion does not hold, proof obligations defined by the set of critical pairs, are produced. For example, in Fig. 4, both axioms Black@ and @Black are obviously redundant since their left-members can be unified on $Black@Black$. The corresponding right members (here $Black$ and $Black$) have to be checked to be interpreted by the same value (what is obvious here).

3) Completeness: As we said previously, completeness is not imposed. Hence, we accept specifications with non isomorphic models. To deal with this situation, unspecified cases of total operations will be supplemented by allocating them to any correct value (see Section IV-B.3). By the confluence property, no inconsistency will be introduced.

D. Predicates

As usual, predicates are considered as total operations with boolean values as expected results. Predicate axioms must then be defined by equivalences (see the Fig. 5):

$$p(u_1, ..., u_n) \Leftrightarrow E$$

where E is a boolean expression, i.e. a first-order formula only built with logical connectives, predicates and equalities (i.e. without quantifier). These equivalences can be easily transformed into defining equations by translating E into a Boolean term. Actually, predefined O'Caml Boolean functions are directly used for this translation (see Section IV).

pred $basic : Color$ %{Tests if thesr only one basic color.}%
$\forall$ *b*, *b1*, *b2*: *Basic*; *c*: *Color*
• $\neg \ basic(Black)$ %(basicBlack)%
• $basic(mix(b, Black))$ %(basicmixBlack)%
• $basic(mix(b1, mix(b2, c))) \Leftrightarrow b1 = b2 \wedge basic(mix(b2, c))$
%(basicmixmix)%

Fig. 5. Predicate

E. Partial operations

In [11], a sub-language based on free extensions is defined for partial operations. Functions are considered as undefined for unspecified input values. This sub-language is very attractive because there is very few constraints. We adapt its principles for basic specifications. Of course, termination is not imposed for partial operations. As previously (see Section III-C.2), confluence is required and proof obligations are generated when it fails. Also completeness is not imposed and specifications are supplemented when they are incomplete.

op $basic : Color \rightarrow ? \ Basic$
%{Returns basic color, if it is single.}%
$\forall$ *b*: *Basic*; *c*: *Color*
• $basic(mix(b, Black)) = b$ %(op_basic_mixBlack)%
• $basic(mix(b, mix(b, c))) = basic(mix(b, c))$ %(opbasicmixmix)%

Fig. 6. Partial operation

Another way in CASL to deal with partial operations is to use definition a predicate introduced by the notation *def* (see Fig. 7). To be handled in our sub-language, the definition predicates must be similar to other predicates (see the Section III-D). Hence, *definition axioms* must have the following form:

$$def \ f(u_1, ..., u_n) \Leftrightarrow E$$

where f is a partial operation, $u_1..., u_n$ are terms built on generators and E is a Boolean expression.

$\forall$ *b*: *Basic*; *c*: *Color*
• $def \ basic(c) \Leftrightarrow basic(c)$ %(opbasicdef)%
%% Definition domain given by the predicate $basic$ (Fig. 5).
• $basic(mix(b, Black)) = b$ %(opbasicmixBlack)%
• $basic(mix(b, mix(b, c))) = basic(mix(b, c))$ %(opbasicmixmix)%

Fig. 7. Definition predicate

The presence of definition axioms can supplement or contradict other equations. For example, according to the axiom %(opbasicdef)% in Fig. 7, the operation $basic$ is undefined for mixtures containing many basic colors. This supplements the two following equations: %(opbasicmixBlack)% and %(opbasicmixmix)%. Indeed, the equality in CASL is strong, it holds if both members are either defined and yield the same value, or it is undefined. Thus, defining equations and definition predicate specify domains of partial operations. To carry out these additional verifications, a second rewriting system, called

rewriting system of definition, is built. The definition axioms are directed from left to right and the rule *def* $f(u_1...,u_n) \rightarrow$ *def* t is associated to every defining equation $f(u_1...,u_n) = t$.

In this rewriting system, the term *def* t is reduced to $true$ or $false$ if t is defined or not defined in all models of the specification. Otherwise, *def* t is unspecified. Thus, the rewriting system produced from the equations must be supplemented in a coherent way with the rewriting system of definition. So, if a term t has a unspecified value, it must be computed (in the generated program) by any value if *def* t is reduced to $true$ and computed as undefined otherwise. As previously, the confluence property of the definition system is forced and it is checked by sufficient syntactic constraint and/or proof obligation.

As it was mentioned in introduction, our goal is to simultaneously use predicates, total and partial operations. Hence, additional verifications are necessary to ensure the consistency of the whole and then generating correct programs. Indeed, using total operations and predicates do not pose any problem, it goes there differently from partial operations. In particular, the use of partial operations in the right member of a defining equation that specifies a total operation or a predicate, must be respectively defined or terminating. These supplementary constraints will give rise to supplementary proof obligations.

IV. Generator implementation

The syntactical analysis of CASL specifications is not carried out by our code generator, but rather by the specialized tool Hets [15]. However, we have to check that specifications belong to the sub-language as defined in Section III.

Syntax of CASL and O'Caml are very close. This makes the code generation easy. However, O'Caml is more constrained than CASL, especially, with respect to type declaration order and function declaration and identifiers building. In the following, various treatments are detailed which have been realized to generate correct O'Caml programs from CASL specifications.

A. Generation of types

free type
Tree ::= *node*(*Color*; *Forest*) ;
Forest ::= *empty* | *cons*(*Tree*; *Forest*)

Fig. 8. Mutually recursive type

In O'Caml, a type must occur after argument types of its constructors or simultaneously when types are mutually recursive. For example, the types $Tree$ and $Forest$ of Fig. 8 must occur simultaneously (because they are mutually recursive) but after the type $Color$ used by them (see Fig. 3), which itself must occur after the type $Basic$.

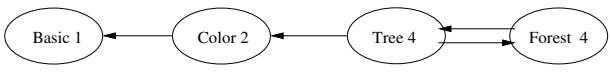

Fig. 9. Dependence graph

To check that this order is respected, a dependence graph is built (see Fig. 9). Each strongly connected component of this graph contains mutually recursive types which will be declared simultaneously.

```
type t_Basic = (* Basic colors *)
    C_Red_Basic | C_Blue_Basic | C_Green_Basic;;

type t_Color = (* Other colors *)
    C_Black_Color
  | C_mix_Basic_Color_Color of t_Basic * t_Color;;

type t_Tree =
    C_node_Basic_Forest_Tree of t_Basic * t_Forest
and t_Forest =
    C_Vide_Forest
  | C_cons_Tree_Forest_Forest of t_Tree * t_Forest;;
```

Fig. 10. Generated code for types

The dependence graph is topologically sorted in order to obtain a generating order preserving the type dependences (see numbering on Fig. 9). From this numbering, the code of Fig. 10 is generated. Note that the comments resulting from the specification are inserted in the corresponding O'Caml code.

B. Generation of functions

As for types, O'Caml functions must occur respecting dependences. Moreover, recursive functions must carry the `rec` note and mutually recursive functions must be simultaneously declared. The situation is thus very close to types. Therefore, as previously, we build a dependence graph that is topologically sorted and strongly connected components are detected inside.

As in Section III, in order to ensure the correctness of generated programs, CASL specifications have to be checked belonging to the CASL sub-language.

1) Termination: For termination property in Section III, it is only imposed that rewriting systems are terminating for total operations and predicates. So right members of definition equations are smaller than left members by using symbol precedence order (due to syntactical conditions). This precedence order is the one defined by dependence graph. Regarding the example of the operation red (Fig. 4), syntactical constraint of termination is preserved.

2) Confluence: In Section III, it was imposed that the resulting rewriting system is confluent. In order that this property holds, left members of defining axioms have not to be pairwise unifiable. When this fails, then the most general unifier gives directly critical pairs that are the proof obligations (see Section III-C.2).

As soon as both termination and/or confluence properties have been checked or proof obligations have been produced, the O'Caml code for operations and predicates is generated. For example, the program given in Fig. 11 has been generated for the total operation red (see Fig. 4).

3) Completeness: As explained in Section III completeness of CASL specifications is not imposed. Nevertheless, the generated O'Caml functions must be complete. For total operations and predicates, this leads to systematically add a default match case (`_ ->`) which returns respectively any data

```
(* Function generated for op red : Color -> Color *)
(* Extract all reds of a color. *)
let rec f_red_Color_Color v_arg1_Color =
  match (v_arg1_Color) with
    (C_Black_Color) ->
      (* Code of the axiom redBlack *)
      C_Black_Color
  | ((C_mix_Basic_Color_Color
        (C_Red_Basic, v_c_Color))) ->
      (* Code of the axiom redRed *)
      (C_mix_Basic_Color_Color
         (C_Red_Basic,
          (f_red_Color_Color v_c_Color)))
  | ((C_mix_Basic_Color_Color
        (C_Blue_Basic, v_c_Color))) ->
      (* Code of the axiom redBlue *)
      (f_red_Color_Color v_c_Color)
  | ((C_mix_Basic_Color_Color
        (C_Green_Basic, v_c_Color))) ->
      (* Code of the axiom redGreen *)
      (f_red_Color_Color v_c_Color)
  | _ -> C_Black_Color ;;
```

Fig. 11. Generated code for functions

(for operations) and `false` (for predicates). Hence, in our example of Fig. 11, a default match case has been added and returns any value of $Color$ sort, here the value $Black$ was chosen. When the operation (or the predicate) are completely specified (as this is the case in our example), this last match case will be simply never performed.

For partial operations, the default match case raises exception `Undef` or returns a value depending on the specification of definition domain.

V. Conclusion and future work

The code generator presented in this article allows to generate both correct and readable O'Caml code for a subclass of CASL specifications. These specifications are basic specifications that can simultaneously contain total and partial operations, and predicates.

Correctness of the produced code depends both on the properties of termination and confluence of the rewriting system associated with equational axioms directed from left to right (see section III). This ensures the existence of a model of the specification, and then its consistency. The system completeness is not imposed. Thus, we can generate code from a specification with many possible models. The code is nevertheless correct and implements one of possible models.

Our generator automatically checks if specification axioms respect some sufficient syntactic criteria. When they hold, the correctness of generated code is automatically guaranteed. Otherwise, the generator produces proof obligations and continues the code generation. The user is supposed to make proofs using tools associated with the CASL language [15]. Of course, this step, which can be long, is not mandatory.

We tried as much as possible to take care over produced code. Hence, translation of identifiers, addition of comments and generation order of various O'Caml types and functions, preserve the user syntactic choices of CASL specifications. This makes the produced code more legible.

Further developments will continue to extend the CASL sublanguage presented here. Especially, conditional axioms must be considered. They are of the form $\alpha_1 \wedge ... \wedge \alpha_m \Rightarrow \alpha$, where $\alpha_1 ... \alpha_m$ are atomic formulas and α is a defining formula as described in section III. To achieve this purpose, we will use results relating to conditional rewriting [16], [17].

CASL structured specifications are not directly handled by the code generator. However this is not really a restriction, since the used Hets analysis tool [15], allows "to flat" most of structured specifications. Nevertheless, it would be necessary to extend our code generator to take into account encapsulation principle (hidden symbols), what can be easily realized by the use of O'Caml modules.

Lastly, our tool is intended for privileged use in the area of geometric modeling. Later, the produced code will be directly interfaced with geometric libraries that offer many usual geometric operations. Thus, the generator will be used for prototyping new operations or geometric algorithms.

Acknowledgments

Special thanks are due to Till Mossakowski for many fruitful and instructive discussions and useful comments.

References

[1] Y. Bertrand, J.-F. Dufourd, J. Françon, and P. Lienhardt, "Algebraic specification and development in geometric modeling," in *TAPSOFT'93*, ser. LNCS, vol. 668, 1993, pp. 75–89.

[2] L. Fuchs, D. Bechmann, Y. Bertrand, and J.-F. Dufourd, "Formal specification for free-form curves and surfaces," in *Spring Conference on Computer Graphics*, Bratislava, Slovakia, 1996.

[3] J.-F. Dufourd, "Algebras and formal specifications in geometric modeling," *Vis. Comp.*, vol. 13, pp. 131–154, 1997, springer-Verlag.

[4] F. Ledoux, A. Arnould, P. L. Gall, and Y. Bertrand, "Geometric modeling with CASL," in *WADT 2001*, ser. LNCS, vol. 2267, 2001, pp. 176–200.

[5] F. Ledoux, J.-M. Mota, A. Arnould, C. Dubois, P. L. Gall, and Y. Bertrand, "Spécifications formelles du chanfreinage," *Tech. et Sci. Info.*, vol. 21, no. 8, pp. 1–26, 2002.

[6] F. Ledoux, "étude et spécifications formelles de l'arrondi d'objets géométriques," Thèse, Université d'Évry Val d'Essonne, décembre 2002.

[7] P. Lienhardt, "Topological models for boundary representation: a comparison with n-dimensional generalized maps," *Computer Aided Design*, vol. 23, no. 1, pp. 59–82, 1991.

[8] L. Fuchs and P. Lienhardt, "Topological structures and free-form spaces," in *Journées de Géométrie Algorithmique*, Barcelone, Spain, 1997.

[9] H. Kirchner and C. Ringeissen, "Executing CASL Equational Specifications with the ELAN Rewrite Engine," LORIA, Nancy (France), Technical Report 99-R-278, 1999.

[10] L. Schröder and T. Mossakowski, "HasCASL: towards integrated specification and development of functional programs," in *AMST'02*, ser. LNCS, vol. 2422, 2002, pp. 99–116.

[11] T. Mossakowski, "Two "Functional Programming" Sublanguages of CASL," CoFI Note L-9, 1998.

[12] A. Arnould, L. Fuchs, M. Aiguier, and T. Brunet, "Génération automatique de code O'Caml à partir de spécifications CASL," Univ. d'Évry, France, Rapport LaMI 113, 2005.

[13] CoFI (The Common Framework Initiative), CASL *Reference Manual*, ser. LNCS 2960 (IFIP Series). Springer, 2004.

[14] F. Baader and T. Nipkow, *Term Rewriting and All That*. C.U. Press, 1998.

[15] U. d. B. Groupe BKB, "Hets - the heterogeneous tool set," http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/.

[16] N. Dershowitz and M. Okada, "A rationale for conditional equational programming," *Th. Comp. Sci.*, vol. 75, pp. 111–138, 1990.

[17] J. Bergstra and J. Klop, "Conditional rewrite rules: Confluence and termination," *J. of Comp. and Syst. Sciences*, vol. 32, pp. 323–362, 1986.