Institution Independent Static Analysis for CASL*

Till Mossakowski[1] and Bartek Klin[2]

[1] BISS, Department of Computer Science, Bremen University
[2] BRICS, Århus University

Abstract. We describe a way to make the static analysis for the in-the-large part of the Common Algebraic Specification Language (CASL) independent of the underlying logic that is used for specification in-the-small. The logic here is formalized as an institution with some extra components. Following the institution independent semantics of CASL in-the-large, we thus get an institution independent static analysis for CASL in-the-large. With this, it is possible to re-use the CASL static analysis for extensions of CASL, or even completely different logics. One only has to provide a static analysis for specifications in-the-small for the given logic. This then can be plugged into the generic static analysis for CASL in-the-large.

1 Introduction

The specification language CASL [Mos97,CoFa,CoF01], based on subsorted partial first-order logic, is the central language of a whole family of languages. Current research includes the development of *extensions* of CASL that also consider the specification of higher-order functions [MHKB00,SM] and of reactive [RR00], [BZ00] and object-oriented [ACZ00] behaviour. Several *restrictions* of CASL to sublanguages [Mosc] make it possible to use specialized tool support.

Connected with this, CASL has a clean separation between specification in-the-small and specification in-the-large. Specification in-the-small here means specification of individual software modules using signatures and axioms, while specification in-the-large concerns the combination of both specifications and software modules. Following this design, also the semantics of CASL exhibits this separation of concerns: The semantics of specification in-the-small is based on a particular institution [GB92], while the semantics of specification in-the-large is defined over an arbitrary but fixed institution [CoFb,Mos00a]. (Strictly speaking, institutions here are replaced by so-called institutions with qualified symbols [Mos00a] in order to admit symbol sets and symbol maps as basic primitives, instead of signature morphisms.) This separation of levels in the semantics makes it possible to re-use the semantics of CASL in-the-large also for the extensions and sublanguages of CASL. Only the semantics of CASL in-the-small has to be adapted individually for each extension (and restricted appropriately for the sublanguages).

* This research was supported by the ESPRIT-funded CoFI Working Group 29432 and by the DFG project MULTIPLE.

Based on the CASL semantics, we have implemented a static analyser for CASL, which is part of the CASL tool set CATS ([Mosa,Mos00b], see also the architecture of CATS in Fig. 1). Now for tool development it is desirable to have a similar separation of levels as in the semantics.

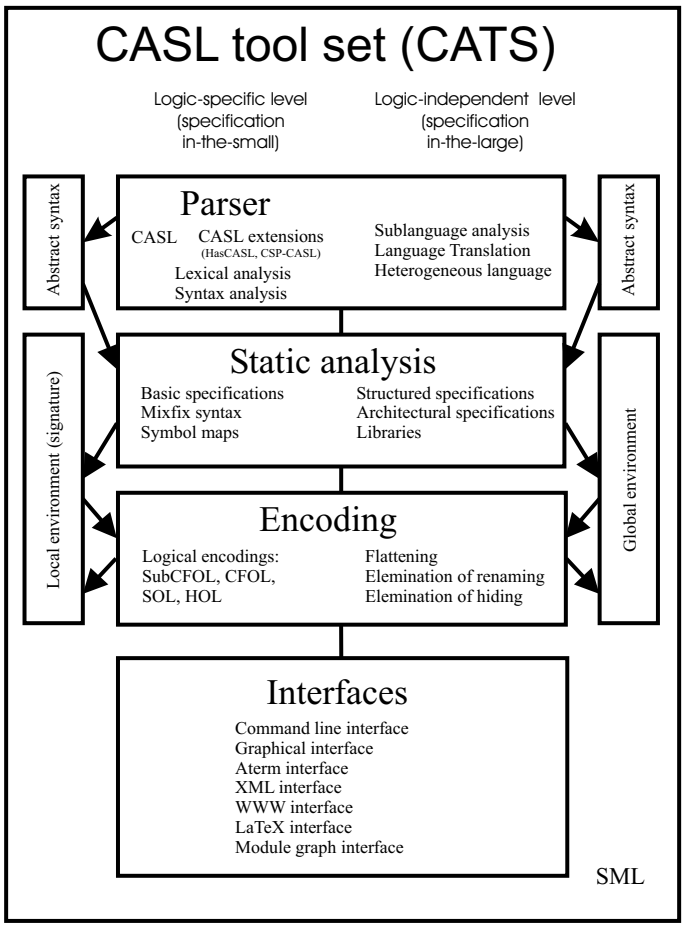


Fig. 1. Architecture of the CASL tool set (CATS)

In this work, we are therefore making our static analysis of CASL in-the-large institution independent, based on the institution independent semantics. This permits a similar economy as for the semantics: the static analysis tools for extensions and sublanguages of CASL need only re-implement the institution-specific part, while the static analysis of CASL in-the-large remains the same for all institutions. As a side-effect, the modular design of the CASL tool set

now becomes a good case study for a CASL architectural specification. Architectural specifications [BST98] are a novel feature of CASL; they allow to describe branching points in system development by indicating units (modules) to be independently developed and showing how these units, once developed, are to be put together to produce the overall result. In the context of the present work, this decomposition roughly will be one into institution-specific and institution independent parts.

The main contribution of this work is not the introduction of new concepts or proof of new results. Rather, it is a bridge between theory and practice. We feel that this is quite an important topic, since it is important both to know whether and how the theoretical concepts really work in practice, and to give a solid theoretical basis for practical implementation work. Along these lines, this paper can also be seen as a step towards bootstrapping CASL by using CASL to develop CASL tools. Of course, much has to be done to achieve this goal, but we feel that such a successful bootstrap would be a very convincing argument in favour of CASL and the methodologies behind it.

The paper is organized as follows: Section 2 recalls the notion of institution with qualified symbols, while section 3 recalls the division of the CASL design into different layers (cleanly separating specification in-the-small from specification in-the-large). Based on this, in section 4, we informally describe how to separate these layers also for the static analysis, thus obtaining a generic static analysis for CASL in-the-large. In section 5, we specify this as a CASL architectural specification. Section 6 contains the conclusions.

2 Preliminaries: Institutions with qualified symbols

The notion of *institution* [GB92] formalizes what a logical system is. The theory of institutions takes a predominantly model-theoretic view of logic, with the satisfaction relation between models and logical sentences adopted as a primary notion. Somewhat unlike in the classical model-theory though, a family of such relations is considered at once, indexed by a category of signatures.

Definition 1. *An* institution $\mathbf{I}$ *consists of:*

- *a category* $\mathbf{Sign}$ *of* signatures*,*
- *a functor* $\mathbf{Sen}\colon \mathbf{Sign} \to \mathbf{Set}$*, giving a set* $\mathbf{Sen}(\Sigma)$ *of* Σ-sentences *for each signature* $\Sigma \in |\mathbf{Sign}|$*,*
- *a functor* $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{Cat}$[1]*, giving a class* $\mathbf{Mod}(\Sigma)$ *of* Σ-models *for each signature* $\Sigma \in |\mathbf{Sign}|$*, and*
- *for* $\Sigma \in |\mathbf{Sign}|$*, a* satisfaction relation $\models_\Sigma \subseteq \mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$*,*

such that for any signature morphism $\sigma\colon \Sigma \to \Sigma'$*,* Σ*-sentence* $\varphi \in \mathbf{Sen}(\Sigma)$ *and* Σ'*-model* $M' \in \mathbf{Mod}(\Sigma')$ *the following* satisfaction condition *holds:*

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \iff \mathbf{Mod}(\sigma)(M') \models_\Sigma \varphi.$$

[1] **Cat** is the (quasi-)category of all categories.

We write $M'|_\sigma$ for $\mathbf{Mod}(\sigma)(M')$, where $\sigma\colon \Sigma \longrightarrow \Sigma' \in \mathbf{Sign}$ and $M' \in \mathbf{Mod}(\Sigma')$.

In the CASL institution ([CoFb]), signatures are the usual many- and subsorted signatures with partial and total operations and predicates, and sentences are formulas of the subsorted partial first order logic with equality and sort generation constraints.

Institutions with qualified symbols add further structure to institutions, mainly to deal with names and qualifications of symbols and with generation of signature morphisms from concise and user-friendly symbol maps. Therefore, an institution with qualified symbols comes along equipped with an underlying set of (fully qualified) symbols, for each signature, and an underlying symbol translation map, for each signature morphism. Moreover, there also is a notion of raw symbols, which includes besides the fully qualified symbols also unqualified or partially qualified symbols as they may be input in specifications.

In order to formalize this, consider an institution $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ additionally equipped with a faithful functor $|\_|\colon \mathbf{Sign} \longrightarrow \mathbf{Set}$ that extracts from each signature the set of (fully qualified) symbols that occur in it (i.e. $(\mathbf{Sign}, |\_|)$ is a concrete category [AHS90]). We assume that there is some fixed 'universe' of *Sym* of symbols that may be used by the specifier when writing symbols (and symbol mappings). In the CASL institution, the natural choice for $|\Sigma|$ is the set of fully qualified symbols of Σ (if we omit qualifications, $|\_|$ is no longer faithful, because symbols may be overloaded with different profiles).

Now in CASL symbol mappings, one may (either partially or completely) omit qualifications of symbols. This leads to the notion of *raw symbol*, which in the case of the CASL institution can be a qualified symbol, an unqualified symbol or a partially qualified symbol. The link between symbols and raw symbols is given by a *matching relation* specifying which symbols correspond to which raw symbols. Finally, in CASL, fitting maps for instantiations of parameterized specifications are automatically extended to compound identifiers, such that they also act on the components. In order to mimick this behaviour within an arbitrary institution, we further assume that there is a set `ID` of *compound identifiers*. This leads to the following definition:

Definition 2 ([Mos00a]). *An* institution with qualified symbols $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models, Sym, |\_|, \mathtt{ID}, RawSym, IDAsRawSym, SymAsRawSym, matches)$ *consists of*

- *an institution* $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$,
- *a set of (fully qualified) symbols* Sym,
- *a faithful functor* $|\_|\colon \mathbf{Sign} \longrightarrow \mathbf{Set}$,
- *a set* `ID` *of compound identifiers,*
- *a set of raw symbols* $RawSym$ *with two injections* $IDAsRawSym\colon \mathtt{ID} \longrightarrow RawSym$ *and* $SymAsRawSym\colon Sym \longrightarrow RawSym$,
- *a matching relation* $matches \subseteq Sym \times RawSym$ *specifying which qualified symbols match which raw symbols,*

such that

– $|\Sigma| \subseteq Sym$ *for each* $\Sigma \in |\mathbf{Sign}|$,

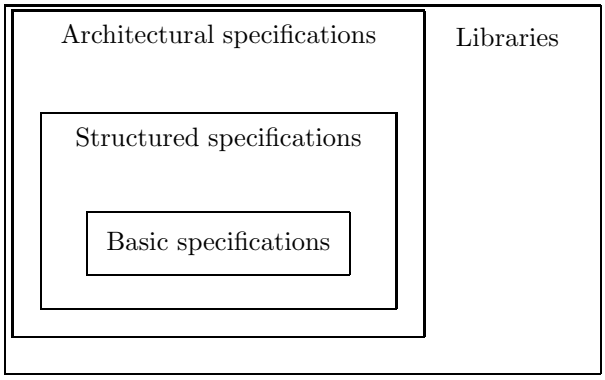
– *for* $id, id_1, \ldots, id_n \in$ `ID`*, also* $id[id_1, \ldots, id_n] \in$ `ID` *(i.e. we can form compound identifiers),*

– SY *matches* $SymAsRawSym(SY')$ *iff* $SY = SY'$ *for* $SY, SY' \in Sym$[2] *and*

– *for each* $SY \in Sym$*, there is a unique* $Ident \in$ `ID` *with* SY *matching* $IDAsRawSym(Ident)$*, called the* name *of* SY*.*

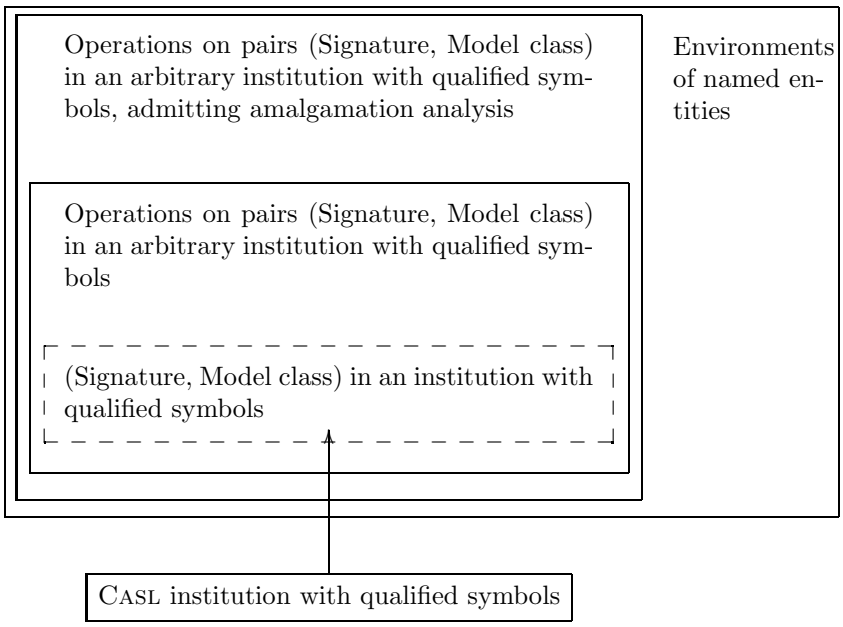
3 The different layers of CASL

The design of CASL has been structured in four different layers, which are largely orthogonal to each other. The first layer, CASL *basic specifications*, allows to formalize axiomatic requirements for a single software module in a specific logic. This corresponds to specification in-the-small. The other three layers are devoted to specification in-the-large: *Structured specifications* allow to combine specifications in a structured way, still refering to specification of *single* software modules. *Architectural specifications* allow to prescribe how to decompose the task of implementing a specification into smaller sub-tasks which can be implemented independently. Finally, basic, structured and architectural specifications can be collected into *libraries*.



The CASL semantic concepts are structured in layers in a similar way:

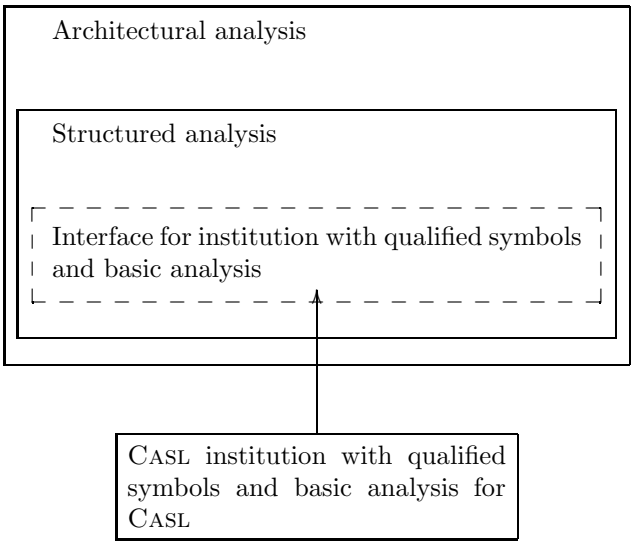
[2] This property is not technically needed in the semantics, but it is desirable since it means that for each ambiguity there is a qualification to resolve it.



The semantics of a CASL basic specification is a signature together with a model class within the CASL *institution with qualified symbols* (cf. Definition 2 above). Now, insofar as the other layers are concerned, the layer of CASL basic specifications can be replaced by any other institution with qualified symbols. More precisely, the semantics of structured specifications can be formalized over an *arbitrary* institution with qualified symbols. To define semantics of architectural specifications, the underlying institution needs to satisfy some additional conditions, connected to the amalgamation property. The CASL institution does not have the amalgamation property, but this problem can be circumvented using an embedding into an enriched institution that has the property, as described in [SMH+01].

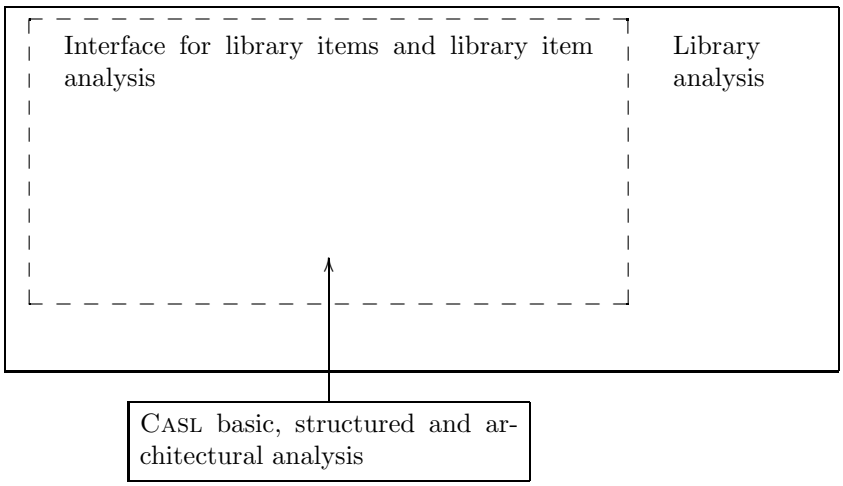
4 Generic static analysis

Now the structure of static analysis follows this layer structuring as well. This enables us to turn the analysis of structured and architectural specifications into a generic program, with the interface consisting of sorts and operations capturing the static part of an institution with qualified symbols (i.e., the signature category and the symbol functor) and, moreover, the static analysis for basic specifications and for symbol maps. Note that symbol maps are usually considered to be part of structured specifications in the CASL documents. However, since they are institution-specific and there is no hope to make them institution-independent, we here count them to the layer of basic specifications.



The generic structured and architectural analysis can then be instantiated with a program module providing the ingredients of the CASL institution with qualified symbols, and a basic analysis for it. Of course, it is now possible to supply any other institution with symbols here.

Note that we have omitted the layer of libraries so far. This has been deliberate, since the analysis of libraries is entirely orthogonal to the rest: we only need to assume that we have a notion of library item and a static analysis for library items. The library analysis then just performs the item analysis for each library item in a given library, and takes care of downloads from other libraries.



The generic library analysis can then be applied to the library item analysis for CASL basic, structured and architectural specifications. Note that we

presently do not have in mind to apply the library analysis within other contexts (although it would be easy to do so). The use of a generic program here mainly has the reason of a clean separation of concerns.

5 Generic static analysis specified in CASL

arch spec LIBRARYANALYSIS =
units I : INSTITUTION;
SAS : INSTITUTION → STRUCTUREDABSTRACTSYNTAX;
AAS : STRUCTUREDABSTRACTSYNTAX → ARCHABSTRACTSYNTAX;
$GEnv$: INSTITUTION → GLOBALENV;
$StrAna$: INSTITUTION×STRUCTUREDABSTRACTSYNTAX×GLOBALENV → STRUCTUREDANALYSIS;
$ArchAna$: INSTITUTION×ARCHABSTRACTSYNTAX×GLOBALENV × STRUCTUREDANALYSIS → ARCHANALYSIS;
$LibAna$: ABSTRACTLIBITEMANALYSIS → LIBANALYSIS;
$SS = SAS[I]$;
$AS = AAS[SS]$;
$G = GEnv[I]$;
$Str = StrAna[I][SS][G]$;
$A = ArchAna[I][AS][G][Str]$;
result
$LibAna\ [A$ **fit** $Entry \mapsto global\_entry,\ Env \mapsto global\_env,$
$analysis \mapsto arch\_analysis]$
end

Fig. 2. Specification of the static analysis of the CASL tool set as a CASL architectural specification

Our task is now to formalize the graphical visualization of the separation of layers in some way. This is exactly the task CASL architectural specifications have been designed for. Hence, an architectural specification of the overall structure of the static analysis is given in Fig. 2.

The unit I, satisfying the specification INSTITUTION (the specification is given in Fig. 3 and Fig. 4 below) contains implementation of all the institution-specific components needed in static analysis of structured and architectural specifications:

- Abstract syntax of basic specifications, symbol maps and symbol lists, conforming to a simple interface specified in the specification BASICABSTRACTSYNTAX (given below),
- a signature category together with a symbol functor and other components of an institution with qualified symbols,
- operations performing basic analysis for the institution, i.e., analysis of basic specifications, symbol maps and symbols lists (note that the analysis of

basic specifications returns a basic specification again, since mixfix grouping analysis might change the initial abstract syntax tree),
- some additional operations, which in principle can be defined over an arbitrary institution with qualified symbols, but only in a very inefficient to compute (if computable at all) way. Hence, to achieve good performance, the task of implementing these operations is assigned to the author of the unit I. These operations include computation of signature union, subsignature relation, induction of signature morphisms from raw symbol maps and a few others,
- an additional operation performing so-called sharing analysis, needed in the static analysis of architectural specifications. In the recently developed diagram semantics of architectural specifications [SMH$^+$01,SMH$^+$], each part of an architectural specification is evaluated to a diagram $D : \mathbf{I} \to \mathbf{Sign}$, where $\mathbf{I}$ is a finite category, and $\mathbf{Sign}$ is the signature category of the underlying institution. A family of models $\langle M_i \rangle_{i \in \mathbf{Ob}(\mathbf{I})}$ is *compatible* with D if for all $i \in \mathbf{Ob}(\mathbf{I})$, $M_i \in \mathbf{Ob}(\mathbf{Mod}(D(i)))$, and for all $m : i \to j$ in $\mathbf{I}$, $M_i = \mathbf{Mod}(D(m))(M_j)$. For two diagrams $D : \mathbf{I} \to \mathbf{Sign}$, $D' : \mathbf{I}' \to \mathbf{Sign}$, where D' extends D, we say that D *ensures amalgamability for* D', if any model family compatible with D can be uniquely extended to a family compatible with D'. We can reduce this to the case where D' extends D by just one object and one morphism into this object.
  We now need a specification of diagrams in the category of signatures and an operation `ensures_amalgamability` checking whether the extension of a diagram with a new node and a new edge (corresponding to some architectural unit term) ensures amalgamability. Further motivation for introducing this operation, together with a description of some difficulties in implementing it in CASL institution, is given in [KHT$^+$,Kli00].

Based on this, several units containing institution independent operations are built:

- SS and AS, based on the syntactic part of I and containing definitions of abstract syntax of structural and architectural specifications,
- G, implementing data structures for global environments needed in the static analysis of structural and architectural specifications,
- Str and A, implementing static analysis of structural and architectural specifications.

The institution independence of these units is expressed by providing methods of constructing them (i.e. parameterized units SAS, AAS, $Genv$, $StrAna$ and $ArchAna$, respectively) that can be applied to any unit satisfying the specification INSTITUTION.

Finally, the resulting unit A is given as a parameter to a parameterized unit $LibAna$, which, given a unit implementing static analysis of some abstract library items (here instantiated with structured and architectural specifications), implements static analysis of libraries of such items.

We now specify an arbitrary institution with qualified symbols in CASL itself. This specification is used as interface specification for the described above (specification of the) generic structured and architectural analysis in Fig. 2.

spec INSTITUTION =
  CATEGORY **with** *object* $\mapsto$ *sign*
and BASICABSTRACTSYNTAX **and** LIST **and** FINITESET **and** FINITEMAP
and DIAGRAM[CATEGORY **with** *object* $\mapsto$ *sign*]
then %% Basic analysis
  ops *basic_analysis* : $BASIC\_SPEC \times sign \rightarrow$
    $BASIC\_SPEC \times sign \times [FORMULA]$;
    stat_symb_map_items : $[SYMB\_MAP\_ITEMS] \rightarrow Map\ raw\_symbol$;
    stat_symb_items : $[SYMB\_ITEMS] \rightarrow [raw\_symbol]$
  %% Structured specifications: Symbols and symbol maps
  sorts $symbol < raw\_symbol$;
    $ID < raw\_symbol$
  ops $||\_||$: $sign \rightarrow Set\ symbol$;
    $||\_||$: $morphism \rightarrow Map\ symbol$;
    $\_ \leq \_$: $pred(symbol \times symbol)$;
    %% Ordering needed for efficient symbol tables
    $\_matches\_$: $pred(symbol \times raw\_symbol)$;
    empty_signature : *sign*;
    $\_[\_]$: $ID \times [ID] \rightarrow ID$;
    name : $symbol \rightarrow ID$
  %% Architectural specifications: sharing analysis
  op *ensures_amalgamability* :
      $diagram \times node \times sign \times edge \times morphism \rightarrow ?\ diagram$
  %% Derived operations
  ops *signature_union* : $sign \times sign \rightarrow ?\ sign$;
    final_union : $sign \times sign \rightarrow ?\ sign$;
    is_subsig : $pred(sign \times sign)$;
    generated_sign, *cogenerated_sign* : $[raw\_symbol] \times sign \rightarrow morphism$;
    induced_from_morphism : $Map\ raw\_symbol \times sign \rightarrow morphism$;
    induced_from_to_morphism : $Map\ raw\_symbol \times sign \times sign \rightarrow morphism$
  %% Axioms omitted
end

Fig. 3. Specification of institutions with qualified symbols

Since our target implementation language is Standard ML [Pau91], a higher-order functional programming language, we use HasCASL [SM,MHKB00], a polymorphic higher-order extension of CASL designed for the specification of Haskell programs, as specification language. In particular, in the examples below, we will use polymorphic type constructors such as $Map\ a$, $Set\ a$, $Table\ a\ b$ and $[a]$ (for maps, sets, index tables and lists). However, note that the use of HasCASL is not really essential here and it is done only to get better readability. The specifica-

spec CATEGORY =
  sorts $object, morphism$
  ops $id : object \to morphism$;
    $\_o\_ : morphism \times morphism \to ? morphism$;
    $dom, cod : morphism \to object$
  %% Axioms omitted
end

spec DIAGRAM[CATEGORY] =
  LIST
then **sorts** $node, edge, diagram$
  ops $dom, cod : edge \to node$;
    $fresh\_node : diagram \to node$;
    $fresh\_edge : diagram \times node \times node \to edge$;
    $add\_node : diagram \times node \times object \to ? diagram$;
    $add\_edge : diagram \times edge \times morphism \to ? diagram$;
    $empty\_diagram : diagram$;
    $object\_at\_node : diagram \times node \to ? object$;
    $morphism\_at\_edge : diagram \times edge \to ? morphism$;
    $diagram\_nodes : diagram \to [node]$;
    $diagram\_edges : diagram \to [edge]$;
    $node\_in\_diagram : pred(node \times diagram)$
  %% Axioms omitted
end

spec BASICABSTRACTSYNTAX =
  %% Minimum needed for interface to structured analysis
  sorts $FORMULA, SYMB\_ITEMS, SYMB\_MAP\_ITEMS, BASIC\_SPEC$
end

Fig. 4. Auxiliary specifications used to specify institutions with qualified symbols

tions can be easily rewritten in plain CASL; but then one has to explicitly include a separate instantiation of lists, maps etc., for each required element type. Also, product sorts have to be introduced explicitly, and operations of predicate type have to be replaced by predicates.

Axioms in this specification are omitted as irrelevant for the general presentation given in this paper. All the needed axioms can be derived from the semantics of CASL.

The specification INSTITUTION, shown in Fig. 3, is essential for extending the CASL tool set CATS with static analysis for future extensions and/or restrictions of CASL. To reuse the static analysis of structured and architectural specifications in an enriched language, one should implement all the specified operations for the new language. If architectural specifications are omitted in the modified language, the implementation of the operation `ensures_amalgamability` can be omitted. For restrictions of CASL, one can either just use the ordinary CASL tool set, combined with a sublanguage analyzer that we have developed, or one can re-implement the analysis of a sublanguage (e.g. to obtain better efficiency) and use the generic analysis for CASL in-the-large as described above.

The specification STRUCTUREDABSTRACTSYNTAX of the abstract syntax for structured specifications is given in Fig. 5. It is based on the abstract syntax for basic specifications (including sorts $SYMB\_ITEMS$ and $SYMB\_MAP\_ITEMS$ as a syntax for raw symbols and raw symbol maps). It follows the CASL abstract syntax [CoF01], except that the productions for $SYMB\_ITEMS$ and $SYMB\_MAP\_ITEMS$ are omitted – they belong to the institution-specific part. We also provide a specification STRUCTUREDANALYSIS with the profiles of the analysis functions for structured specifications. Moreover, the corresponding specifications for the architectural level are given in Fig. 6.

The specification GLOBALENV, given in Fig 7, provides data structures for the information that is extracted by the static analysis. For structured specifications, the structure is roughly kept (mainly, symbol maps are replaced by the corresponding signature morphisms), while basic specifications are fully expanded to signature and set of axioms that they denote. The data structures for architectural specifications closely follow their semantics, as given in [CoFb].

The specification ABSTRACTLIBITEMANALYSIS is given in Fig. 8. It contains a sort LIB_ITEM for the syntax of abstract library items (the constituents of a library), a sort Env for the semantics of these, and a sort $ITEM\_NAME$ to name them. Environments (sort Env) are then just tables of entries indexed by item names. The function *analysis* takes a LIB_ITEM and analysis it in the given environment.

The specification LIBANALYSIS is built on top of ABSTRACTLIBITEMANALYSIS. It provides data structures for both libraries with possible downloads from other libraries (this follows the CASL abstract syntax, except that an intermediate non-terminal LIB_ITEM' is needed here to separate the library item-specific from the library item-independent parts), and a function *check* that statically analyses a library, using the given function *analysis* from ABSTRACTLIBITEMANALYSIS. It returns an environment corresponding to the library, and also takes

spec StructuredAbstractSyntax =
Institution **and** List **and** FiniteMap **and** String
then **sorts** $SPEC\_NAME = string$;
$VIEW\_NAME = string$

free types $FIT\_ARG ::= fit\_spec(SPEC;\ [SYMB\_MAP\_ITEMS])$
$\mid\ fit\_view(VIEW\_NAME;\ [FIT\_ARG])$;

$RENAMING ::= renaming([SYMB\_MAP\_ITEMS])$;
$RESTRICTION ::= hide\_spec([SYMB\_ITEMS])$
$\mid\ reveal\_spec([SYMB\_MAP\_ITEMS])$;

$GENERICITY ::= genericity(PARAMS;\ IMPORTS)$;
$PARAMS ::= params([SPEC])$;
$IMPORTS ::= imports([SPEC])$;

$SPEC ::= basic(BASIC\_SPEC)$
$\mid\ translation(SPEC;\ RENAMING)$
$\mid\ reduction(SPEC;\ RESTRICTION)$
$\mid\ union\_spec([SPEC])$
$\mid\ extension([SPEC])$
$\mid\ free\_spec(SPEC)$
$\mid\ local\_spec(SPEC;\ SPEC)$
$\mid\ closed\_spec(SPEC)$
$\mid\ spec\_inst(SPEC\_NAME;\ [FIT\_ARG])$;

$VIEW\_TYPE ::= view\_type(SPEC;\ SPEC)$;

$PRE\_LIB\_ITEM ::= spec\_defn(SPEC\_NAME;\ GENERICITY;\ SPEC)$
$\mid\ view\_defn(VIEW\_NAME;\ GENERICITY;\ VIEW\_TYPE)$

end

spec StructuredAnalysis =
StructuredAbstractSyntax **and** GlobalEnv
then
ops $spec\_analysis : sign \times global\_env \times SPEC \to ? spec\_lenv \times SPEC$;
$structured\_analysis :$
$global\_env \times PRE\_LIB\_ITEM \to ? global\_env \times PRE\_LIB\_ITEM$
%% Axioms and hidden operations omitted
end

Fig. 5. Specification of structured abstract syntax and analysis

spec ArchAbstractSyntax =
StructuredAbstractSyntax
then
sorts *UNIT_NAME* = *string*;
ARCH_SPEC_NAME = *string*;
UNIT_TYPE_NAME = *string*

free types

ARCH_SPEC ::= *basic_arch_spec*([*UNIT_DECL_DEFN*]; *RESULT_UNIT*)
| *named_arch_spec*(*ARCH_SPEC_NAME*);

UNIT_DECL_DEFN ::= *unit_decl_case*(*UNIT_DECL*)
| *unit_defn_case*(*UNIT_DEFN*);
UNIT_DECL ::= *unit_decl*(*UNIT_NAME*; *UNIT_SPEC*; *UNIT_IMPORTED*);
UNIT_IMPORTED ::= *unit_imported*([*UNIT_TERM*]);
UNIT_DEFN ::= *unit_defn*(*UNIT_NAME*; *UNIT_EXPRESSION*);

UNIT_SPEC ::= *unit_type_case*(*UNIT_TYPE*)
| *spec_name_case*(*SPEC_NAME*)
| *arch_spec_case*(*ARCH_SPEC*)
| *closed_unit_spec*(*UNIT_SPEC*);
UNIT_TYPE ::= *unit_type*([*SPEC*]; *SPEC*);

RESULT_UNIT ::= *result_unit*(*UNIT_EXPRESSION*);
UNIT_EXPRESSION ::= *unit_expression*([*UNIT_BINDING*]; *UNIT_TERM*);
UNIT_BINDING ::= *unit_binding*(*UNIT_NAME*; *UNIT_SPEC*);
UNIT_TERM ::= *unit_translation*(*UNIT_TERM*; *RENAMING*)
| *unit_reduction*(*UNIT_TERM*; *RESTRICTION*)
| *amalgamation*([*UNIT_TERM*])
| *local_unit*([*UNIT_DEFN*]; *UNIT_TERM*)
| *unit_appl*(*UNIT_NAME*; [*FIT_ARG_UNIT*]);
FIT_ARG_UNIT ::= *fit_arg_unit*(*UNIT_TERM*; [*SYMB_MAP_ITEMS*]);

LIB_ITEM ::= **sort** *PRE_LIB_ITEM*
| *arch_spec_defn*(*ARCH_SPEC_NAME*; *ARCH_SPEC*)
| *unit_spec_defn*(*SPEC_NAME*; *UNIT_SPEC*)
end

spec ArchAnalysis =
ArchAbstractSyntax **and** StructuredAnalysis
then
op *arch_analysis* : *global_env* × *LIB_ITEM* →? *global_env* × *LIB_ITEM*
%% Axioms and hidden operations omitted
end

Fig. 6. Specification of architectural abstract syntax and analysis

```
spec  GlobalEnv =
      Institution and List and Table and String
then
sorts SPEC_NAME = string;
      UNIT_NAME = string;
      ITEM_NAME = string
free types
      spec_env ::= basic_env(sign; [FORMULA])
                 | translate_env(spec_env; morphism)
                 | derive_env(spec_env; morphism)
                 | union_env([spec_env])
                 | extension_env([spec_env])
                 | free_spec_env(spec_env)
                 | closed_spec_env(spec_env)
                 | spec_inst_env(SPEC_NAME; spec_env; morphism; [spec_env]);
      %% intended use: (name, body, fitting morphism, actual args)
      spec_lenv ::= SPEC_ENV(sign; sign; spec_env)
      %% intended use: (flattened sign, flattened hidden sign, env)
op    empty_spec_lenv : spec_lenv
type  genericity_env = (spec_lenv × [spec_lenv] × sign)
      %% intended use: (union of envs for all imports,
      %%       list of envs for formal parameters,
      %%       signature union of all imports and all formal parameters)
type  comp_sigs = [sign];
      unit_sig = comp_sigs × sign;
      st_based_unit_ctx = Table UNIT_NAME node;
      based_par_unit_sig = node × par_unit_sig;
      st_par_unit_ctx = Table UNIT_NAME based_par_unit_sig;
      ext_st_unit_ctx = st_par_unit_ctx × st_based_unit_ctx × diagram;
      arch_sig = ext_st_unit_ctx × unit_sig
      %% architectural signatures
free type global_entry ::= spec_defn_env(genericity_env; spec_lenv)
                         | view_defn_env(genericity_env; spec_lenv; morphism; spec_lenv)
                         | arch_spec_defn_env(arch_sig)
                         | unit_spec_defn_env(unit_sig)
type  global_env = Table ITEM_NAME global_entry
op    empty_global_env : global_env
end
```

Fig. 7. Specification of global environments

```
spec  AbstractLibItemAnalysis =
      String and Table
then  sorts  LIB_ITEM, Entry;
             ITEM_NAME = string;
             Env = Table ITEM_NAME Entry
      ops    analysis : Env × LIB_ITEM →? Env × LIB_ITEM
end

spec  LibAnalysis =
      AbstractLibItemAnalysis
then
sorts LIB_NAME = string
free  types
      ITEM_NAME_OR_MAP ::= item_name(ITEM_NAME)
                         | item_name_map(ITEM_NAME; ITEM_NAME);
      LIB_ITEM'        ::= sort LIB_ITEM
                         | download_items(LIB_NAME;
                                          [ITEM_NAME_OR_MAP]);
      LIB_DEFN         ::= lib_defn(LIB_NAME × [LIB_ITEM]);
      Lib_Env          ::= lib_env(Table LIB_NAME (Env × LIB_DEFN))
op    check : Lib_Env × LIB_DEFN → Lib_Env × Env × LIB_DEFN
      %% Axioms and hidden operations omitted
end
```

Fig. 8. Specification of abstract library analysis

and returns a library environment, which is a table associating library names with environments.

6 Conclusion and future work

We have shown how to structure the static analysis of CASL within the CASL tool set (CATS) in a modular way. We therefore have followed the structuring of the CASL design and semantics into several layers, and outlined how the static analysis can be turned into a generic program which is parameterized over an arbitrary institution (plus some extra components). We also have specified this modular structure of the CASL static analysis in CASL itself, using a CASL architectural specification with parameterized units.

At the implementation level, these parameterized units are realized as Standard ML functors (SML is the implementation language of the CASL tool set). Thus, this work can also be seen as a case study of relating CASL architectural specifications and Standard ML implementations. The size of the different components of the CASL tool set is as follows:

Parsing and printing	7.000 lines of code
Institution-specific analysis	9.000 lines of code
Institution-independent analysis	9.000 lines of code
Total	25.000 lines of code

This means that for replacing the CASL institution with some other institution like higher-order CASL, one has to re-program both the institution-specific analysis and the parsing and printing, while the generic institution-independent analysis can be re-used.

In the future, this should be applied to several extensions of CASL, like HO-CASL [MHKB00], HasCASL [SM], CASL-LTL [RR00], SB-CASL [BZ00], LB-CASL [ACZ00], etc.

We hope that also large parts of the institution-specific analysis of CASL basic specifications can be re-used, since these extensions are built on top of CASL. However, it is not clear whether this re-use can be turned into a generic program. At the moment, this kind of re-use seems to be more a "copy and paste" re-use. The same also holds for parsing and printing, since generally it seems to be difficult to build parsers for "grammars with holes" and to instantiate the holes with different (institution-specific) grammars later on.

A related direction of future work is the construction of a static analysis for heterogeneous CASL [Mosb]. Heterogeneous CASL combines several logics for basic specifications within one language, while the structured and architectural specifications basically are those of CASL.

References

[ACZ00] D. Ancona, M. Cerioli, and E. Zucca. Extending CASL by late binding. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[AHS90] J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Wiley, New York, 1990.

[BST98] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *AMAST '98, Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology, Manaus*, volume 1548 of *LNCS*, pages 341–357. Springer-Verlag, 1998.

[BZ00] H. Baumeister and A. Zamulin. State-based extension of CASL. In *Proceedings IFM 2000*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[CoFa] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

[CoFb] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (Documents/CASL/Semantics, version 1.0), in [CoFa], forthcoming.

[CoF01] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [CoFa], March 2001.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: LNCS 164, 221–256, 1984.

[KHT+] B. Klin, P. Hoffman, A. Tarlecki, T. Mossakowski, and L. Schröder. Checking amalgamability conditions for CASL architectural specifications. In J. Sgall, A. Pultr, P. Kolman, editors, *Mathematical Foundations of Computer Science*, volume 2136 of *Lecture Notes in Computer Science*, pages 451–463. Springer-Verlag, 2001.

[Kli00] B. Klin. An implementation of static semantics for architectural specifications in CASL (in Polish). Master's thesis, Warsaw University, 2000.

[MHKB00] T. Mossakowski, A. Haxthausen, and B. Krieg-Brückner. Subsorted partial higher-order logic as an extension of CASL. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 126–145. Springer-Verlag, 2000.

[Mosa] T. Mossakowski. The CASL tool set. Available at `http://www.tzi.de/cofi/CATS`.

[Mosb] T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. Submitted.

[Mosc] Till Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*. To appear.

[Mos97] Peter D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97, Proc. Intl. Symp. on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997.

[Mos00a] T. Mossakowski. Specification in an arbitrary institution with symbols. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 252–270. Springer-Verlag, 2000.

[Mos00b] Till Mossakowski. CASL: From semantics to tools. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 93–108. Springer-Verlag, 2000.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[RR00] G. Reggio and L. Repetto. CASL-CHART: a combination of statecharts and of the algebraic specification language CASL. In *Proc. AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[SM] L. Schröder and T. Mossakowski. HasCASL: Towards integrated specification and development of Haskell programs. Submitted.

[SMH+01] L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, and A. Tarlecki. Semantics of architectural specifications in CASL. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 253–268. Springer-Verlag, 2001.

[SMH+] L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, and A. Tarlecki. Amalgamation in the semantics of CASL. Submitted to Theoretical Computer Science.