11

Tools

Till Mossakowski

This chapter gives an overview of the CASL tools. Analysis tools for CASL like parsers and static checkers, as well as formatters, are stable now and cover the whole of CASL. Proof tools are available but are less mature.

CASL has been designed with the goal of subsuming many previous specification languages. Most of these languages come with specific tools, and of course, these tools should be reusable in the context of CASL. Hence, a central issue is to build bridges to existing tools (rather than building new tools from scratch). Using an interchange format generated by the analysis tools, CASL has been interfaced in this way to rewriting engines and theorem provers, usually working for a subset of CASL.

Naturally, due to the ongoing development of these tools, detailed descriptions would become outdated sooner or later. Therefore, we give here just an appetizer, intended to encourage the reader to install the tools and experiment with them (and to convince her/him that this is rather easy). More detailed descriptions of the tools, as well as their latest versions and other tools that may be developed in the future, are available by following the links on the CoFI tools home page [21]: `http://www.cofi.info/Tools`.

The analysis tools for CASL have been used to check all the examples contained in this book, as well as the CASL Basic Libraries [20]. Moreover, some proofs from a case study in refinement have been carried out with the proof tools.

CASL specifications can be checked for well-formedness using a form-based web page.

The easiest way to check a CASL specification for well-formedness is to visit the web interface. Using a web form, you can submit your specification (without the need to install anything on your machine), and get immediate

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 131–142, 2004.

feedback about the well-formedness of the specification. Follow the "web interface" link on the CoFI tools home page.

11.1 The Heterogeneous Tool Set (HETS)

The Heterogeneous Tool Set (HETS) is the main analysis tool for CASL.

HETS is a tool set for the analysis of specifications written in CASL, its extensions and sublanguages – hence the name "heterogeneous". HETS consists of logic-specific tools for the parsing and static analysis of the different CASL extensions and sublanguages, as well as a logic-independent parsing and static analysis tool for structured and architectural specifications and libraries (which of course calls the logic-specific tools whenever a basic specification is encountered). In order to be able to tackle proof obligations occurring in (statically well-formed) specifications, HETS is interfaced with various logic-specific theorem proving, rewriting and consistency checking tools. On top of this, there is a logic-independent proof engine called MAYA, which manages the proof obligations. MAYA uses so-called *development graphs*, a graphical representation of CASL structured specifications.

The architecture of HETS is shown in Fig. 11.1. The latest version can be obtained from the CoFI tools home page [21]. Installation is easy; just follow the instructions.

Consider the first example in this book:

spec STRICT_PARTIAL_ORDER =
  sort *Elem*
  pred $\_\_ < \_\_ : Elem \times Elem$
  $\forall x, y, z : Elem$
  - $\neg(x < x)$ %(strict)%
  - $x < y \Rightarrow \neg(y < x)$ %(asymmetric)%
  - $x < y \wedge y < z \Rightarrow x < z$ %(transitive)%

  %{ Note that there may exist x, y such that neither $x < y$ nor $y < x$. **}%**

end

HETS can be used for parsing and checking static well-formedness of specifications.

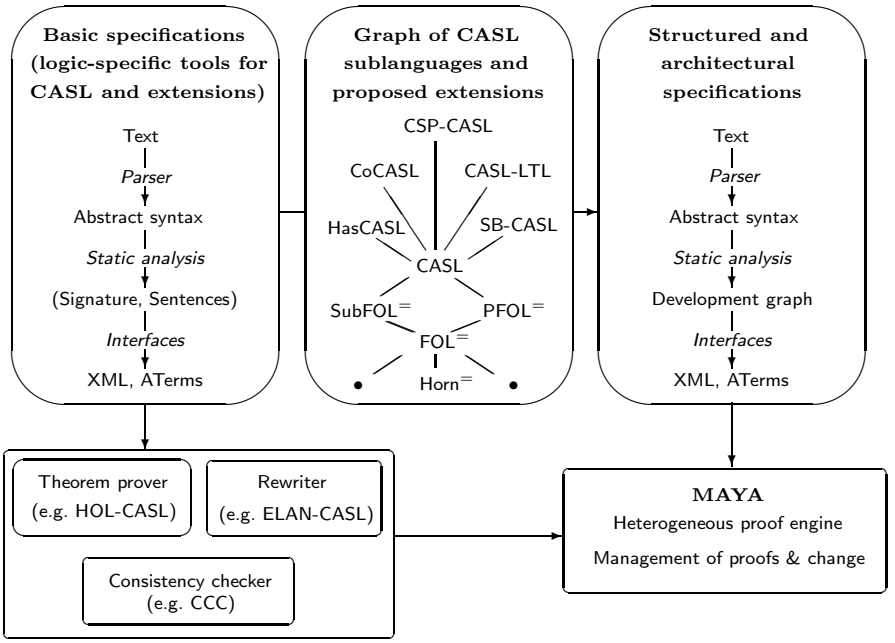

Fig. 11.1. Architecture of the heterogeneous tool set.

Let us assume that the example is in a file named `Order.casl` (actually, this file is provided on the web and on the CD-ROM coming with this volume). Then you can check the well-formedness of the specification by typing (into some shell):

```
hets Order.casl
```

HETS checks both the correctness of this specification with respect to the CASL syntax, as well as its correctness with respect to the static semantics (e.g. whether all identifiers have been declared before they are used, whether operators are applied to arguments of the correct sorts, whether the use of overloaded symbols is unambiguous, and so on).

It is also possible to generate a pretty printed LaTeX version of `Order.casl` by typing:

```
hets -o pp.tex Order.casl
```

One use of `Order.casl` might be to express the fact that the natural numbers form a strict partial order as a view, as follows:

spec NATURAL = **free type** $Nat ::= 0 \mid suc(Nat)$ **end**

spec NATURAL_ORDER_2 =
NATURAL
then **pred** $\_\_ < \_\_ : Nat \times Nat$
$\forall x, y : Nat$
- $0 < suc(x)$
- $\neg x < 0$
- $suc(x) < suc(y) \Leftrightarrow x < y$

end

view V1 : STRICT_PARTIAL_ORDER **to** NATURAL_ORDER_2 =
$Elem \mapsto Nat$

end

Again, these specifications can be checked with HETS. However, this only checks syntactic and static semantic well-formedness – it is *not* checked whether the predicate '$\_\_ < \_\_$' introduced in NATURAL_ORDER_2 actually is constrained to be interpreted by a strict partial ordering relation. Checking this requires theorem proving, which will be discussed below.

> *HETS also displays and manages proof obligations, using development graphs.*

However, before coming to theorem proving, let us first inspect the proof obligations arising from a specification. This can be done with:

```
hets -g Order.casl
```

(assuming that the above two specifications and the view have been added to the file `Order.casl`). HETS now displays a so-called development graph (which is just an overview graph showing the overall structure of the specifications in the library), see Fig. 11.2.

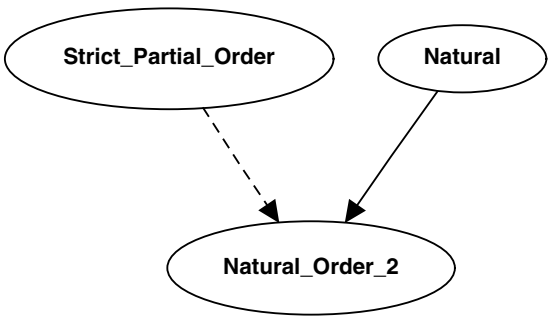

Fig. 11.2. Sample development graph.

Nodes in a development graph correspond to CASL specifications. Arrows show how specifications are related by the structuring constructs.

The solid arrow denotes an ordinary import of specifications (generated by the extension), while the dashed[1] arrow denotes a proof obligation (corresponding to the view). This proof obligation needs to be discharged in order to show that the view is well-formed.

As a more complex example, consider the following loose specification of a sorting function, taken from Chap. 6:

spec LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_ < \_$] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$(cons(e', L)\ is\_sorted \wedge \neg(e' < e))$
within **op** $order : List \to List$
$\forall L : List \bullet order(L)\ is\_sorted$
end

The following specification of sorting by insertion also is taken from Chap. 6:

spec LIST_ORDER [TOTAL_ORDER **with sort** *Elem*, **pred** $\_\_ < \_\_$] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **op** $insert : Elem \times List \to List$
$\forall e, e' : Elem;\ L : List$
• $insert(e, empty) = cons(e, empty)$
• $insert(e, cons(e', L)) = cons(e', insert(e, L))\ when\ e' < e$
$else\ cons(e, cons(e', L))$
within **op** $order : List \to List$
$\forall e : Elem;\ L : List$
• $order(empty) = empty$
• $order(cons(e, L)) = insert(e, order(L))$
end

Both specifications are related. To see this, we first inspect their signatures. This is possible with:

```
hets -g Sorting.casl
```

[1] Actually, the dashed arrow will be displayed as solid and in red by HETS; we do not have colors available here.

assuming that `Sorting.casl` contains the above specifications. HETS now displays a more complex development graph, see Fig. 11.3.

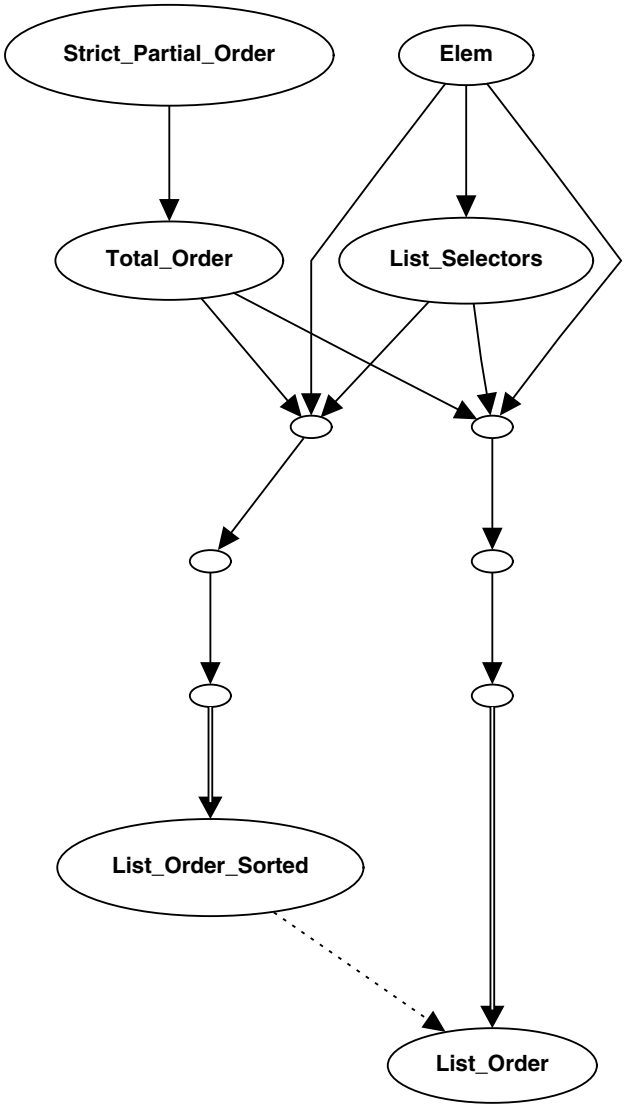

Fig. 11.3. Development graph for the two sorting specifications.

Internal nodes in a development graph correspond to unnamed parts of a structured specification.

In the above-mentioned development graph, we have two types of nodes. The named ones correspond to named specifications, but there are also unnamed nodes corresponding to anonymous basic specifications like the declaration of the *insert* operation in LIST_ORDER above. Basically, there is an internal node for each structured specification that is not named.

Again, the simple solid arrows denote an ordinary import of specifications (corresponding to the extensions and unions in the specifications), while the double arrows denote hiding (corresponding to the local specification).

By clicking on the nodes, one can inspect their signatures. In this way, we can see that both LIST_ORDER_SORTED and LIST_ORDER have the same signature. Hence, it is legal to add a view:

view V2[TOTAL_ORDER] : LIST_ORDER_SORTED[TOTAL_ORDER] **to** LIST_ORDER[TOTAL_ORDER]
end

We have already added this view to `Sorting.casl`. The corresponding proof obligation between LIST_ORDER_SORTED and LIST_ORDER is displayed in Fig. 11.3 as a dotted arrow.

Proof obligations can be discharged in various ways.

Trivial proof obligations can be discharged by HETS alone using the "Proofs" menu. The proof obligation in Fig. 11.3, indicated by the lower dotted arrow between LIST_ORDER_SORTED and LIST_ORDER, states that insertion sort, as defined by the operation *order* in LIST_ORDER, actually has the properties of a sorting algorithm. Here, one has to choose a theorem prover that is to be used to discharge the proof obligation, which is then done by using commands specific to the theorem prover (cf. e.g. Section 11.2). Alternatively, one can state that one just conjectures the obligation to be true.

HETS is written in Haskell. Its parser uses combinator parsing. The user-defined (also known as "mixfix") syntax of CASL calls for a two-pass approach. In the first pass, the skeleton of a CASL abstract syntax tree is derived, in order to extract user-defined syntax rules. In a second pass, which is performed during static analysis, these syntax rules are used to parse any expressions that use mixfix notation. The output is stored in the so-called ATerm format [12], which is used as interchange format for interfacing with other tools.

HETS provides an abstract interface for institutions, so that new logics can be integrated smoothly. In order to do so, a parser, a static checker and a prover for basic specifications in the logic have to be provided.

Hets has been built based on experiences with its precursors, Cats and Maya. The Casl Tool Set (Cats) [28, 30] comes with roughly the same analysis tools as Hets. The management of development graphs is not integrated in Cats, but is provided with a stand-alone version of the tool Maya [5, 4]. Cats and Maya can be obtained from the CoFI tools home page [21].

11.2 Hol-Casl

Hol-Casl is an interactive theorem prover for Casl, based on the tactical theorem prover Isabelle.

The Hol-Casl system [28] provides an interface between Casl and the theorem proving system Isabelle/Hol [31]. We have chosen Isabelle because it has a very small core guaranteeing correctness, and its provers, like the simplifier or the tableaux prover, are built on top of this core. Furthermore, there is over ten years of experience with it, and several mathematical textbooks have been partially verified with Isabelle.

Casl is linked to Isabelle/Hol by an encoding.

Since subsorting and partiality are present in Casl but not in Isabelle/Hol, we have to encode these features, as explained in [29]. This means that theorem proving is not done in the Casl logic directly, but in the logic Hol (for higher-order logic) of Isabelle. Hol-Casl tries to make the user's life easy by:

- choosing a shallow embedding of Casl into Hol, which means that e.g. Casl's logical implication `=>` is mapped directly to Isabelle/Hol's logical implication `-->` (and the same holds for other logical connectives and quantifiers); and
- adapting Isabelle/Hol's syntax to conform with the Casl syntax, e.g. Isabelle/Hol's `-->` is displayed as `=>`, as in Casl.

However, it is essential to be aware of the fact that the Isabelle/Hol logic is different from the Casl logic. Therefore, the formulas appearing in subgoals of proofs with Hol-Casl will not fully conform to the Casl syntax: they may use features of Isabelle/Hol such as higher-order functions that are not present in Casl. Hol-Casl can be obtained from the CoFI tools home page [21].

To start the Hol-Casl system, follow the installation instructions, and then type:

```
HOL-CASL
```

You can load the above specification file `Order.casl` by typing:

```
use_casl "Order";
```

Let us try to prove part of the view V1 above. To prepare for conducting a proof in the target specification of the view, NATURAL_ORDER_2, type in:

```
CASL_context Natural_Order_2.casl;

AddsimpAll();
```

The first command just selects the specification as the current proof context; the second one adds all the axioms of the specification to ISABELLE's simplifier (a rewriting engine). Note that the latter is advisable only if the axioms are terminating, when considered as a set of rewrite rules.

To prove the first property expressed by the view, we first have to type in the goal. Then we chose to perform induction over the variable x, and the rest can be done with automatic simplification. Finally, we name the theorem for later reference:

```
Goal "forall x:Nat . not x<x";
by (induct_tac "x" 1);
by Auto_tac;
qed "Nat_irreflexive";
```

Both the stand-alone MAYA as well as the MAYA part of HETS also provide an interface to HOL-CASL, so that it can be used to discharge proof obligations arising in development graphs.

11.3 ASF+SDF Parser and Syntax-Directed Editor

ASF+SDF was used to prototype the CASL syntax.

The algebraic specification formalism ASF+SDF [22] and the ASF+SDF Meta-Environment [13] have been deployed to prototype CASL's concrete syntax, and to develop a mapping for the concrete syntax to an abstract syntax representation using so-called ATerms [12]. Currently, only the first pass of parsing (i.e. without mixfix analysis) is realized in SDF. Parsing is performed based on the underlying Scannerless Generalized LR parsing technology. A prototype of the mapping from the concrete to abstract representation is written in ASF rewrite rules.

The ASF+SDF Meta-Environment provides syntax-directed editing of CASL specifications.

Given the concrete syntax definition of CASL in SDF, syntax-directed editors within the ASF+SDF Meta-Environment come for free. Recent developments in the Meta-Environment even allow for the development of a CASL specification environment.

The ASF+SDF Meta-Environment provides a built-in library mechanism which contains a collection of grammars, among others CASL. Via the library the user of the Meta-Environment has access to the CASL syntax in SDF and a collection of ASF equations to map CASL specifications into an interchange format named CasFix. The `asfc` tool compiles the CASL grammar into a stand-alone C program.

A link to the ASF+SDF Meta-Environment with further information and a download possibility can be found at the CoFI tools home page [21]. The built-in library of the Meta-Environment (Version 1.5 and higher) will provide the full CASL grammar in SDF and the mapping to CasFix as an ASF specification.

11.4 Other Tools

The following tools are at a prototype stage at the time of appearance of this volume. Please refer to at the CoFI tools home page [21], where the latest versions can be downloaded.

Translation to OCAML

A translation from CASL into OCAML has been developed at Paris and Poitiers. The translation works for a "functional programming" sublanguage of CASL that includes free datatypes and recursive definitions of operations over these types.

Translation to Haskell

A translation from CASL into Haskell has been developed in Bremen. Actually, the translation works on an executable subset of HASCASL (a higher-order extension of CASL). Using an embedding of CASL into HASCASL, this translation can also be used for CASL.

ELAN-CASL

ELAN-CASL is a rewriting engine for the HORN$^{=}$ sublanguage of CASL. It is based on a translation of that sublanguage to the input language of the rewriting engine ELAN.

Given a CASL specification and a query term, ELAN-CASL computes the normal forms of the term. 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.

ELAN can be called either as an interpreter or as a compiler.

CASL consistency checker

The CASL consistency checker (CCC) has been developed in Bremen and Swansea. With CCC, one can interactively check whether a CASL specification is consistent. It is a faithful implementation of a consistency calculus for full CASL [32].

Besides using certain syntactical criteria, the consistency calculus relies heavily on the CASL structuring mechanisms and their semantic annotations. Consequently, consistency proofs follow the structure of the given specification. In this way, the calculus highlights the (usually few) 'hot spots' of a specification (e.g. views requiring theorem proving), while the (lengthy) 'trivial' parts of the consistency argument are discharged automatically. As the consistency calculus works along the structure of the specification, the need to construct (and prove correct) actual models of specifications is avoided as far as possible.

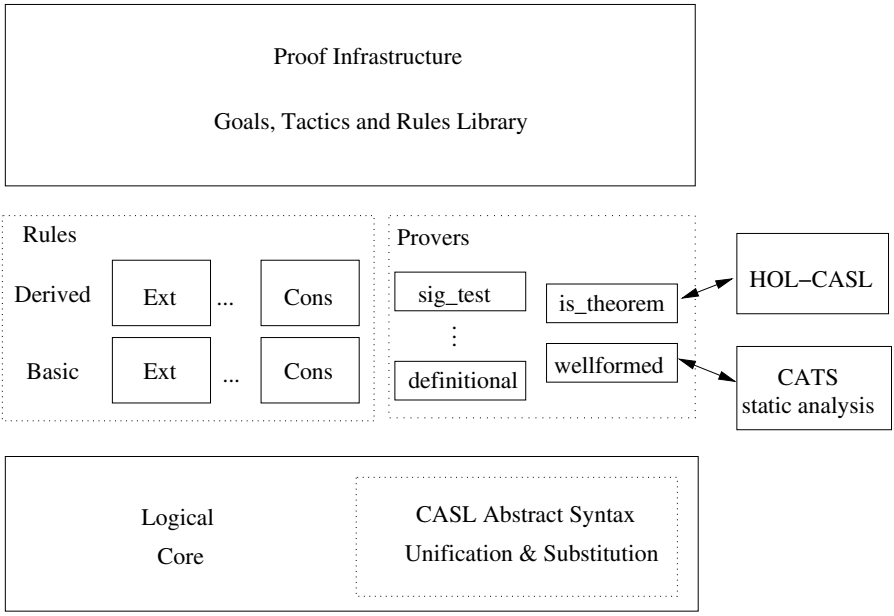

Fig. 11.4. CCC System Architecture

The CASL Consistency Checker consists of four parts (see Fig. 11.4): first, a *logical core* implementing a representation of the propositions to be proved, and the basic proof rules, along with a generic unification package for the CASL abstract syntax; secondly, representations of the *calculi*: the base rules which are stated axiomatically, and derived rules; thirdly, *proof procedures* (automatic or semi-automatic decision procedures), which serve to discharge specific proof obligations; and finally, *proof infrastructure* such as a package which helps users to conduct goal-directed proofs, tactics that support writing advanced proof procedures, and a simple database which allows storage and retrieval of proved theorems. The system is encapsulated by a single interface which does not allow the end-user (i.e. the working specifier) to add any more axiomatic rules or provers; thus, the typing is used to both increase confidence in the correctness of the implementation in the style of LCF, and to protect the integrity of the system from user intervention.