CASL: From Semantics to Tools

Till Mossakowski

Department of Computer Science and Bremen Institute for Safe Systems,
Universität Bremen, P.O. Box 330440, D-28334 Bremen
till@informatik.uni-bremen.de

Abstract. CASL, the common algebraic specification language, has been developed as a language that subsumes many previous algebraic specification frameworks and also provides tool interoperability. CASL is a complex language with a complete formal semantics. It is therefore a challenge to build good tools for CASL. In this work, we present and discuss the Bremen HOL-CASL system, which provides parsing, static checking, conversion to LaTeX and theorem proving for CASL specifications. To make tool construction manageable, we have followed some guidelines: re-use of existing tools, interoperability of tools developed at different sites, and construction of generic tools that can be used for several languages. We describe the structure of and the experiences with our tool and discuss how the guidelines work in practice.

1 Introduction

During the past decades a large number of algebraic specification languages have been developed. Unfortunately, these languages are based on a diversity of basic algebraic specification concepts. The presence of so many similar specification languages with no common framework had hindered the dissemination and application of research results in algebraic specification. In particular, it had made it difficult to produce educational material, to re-use tools and to get algebraic methods adopted in industry. Therefore, in 1995, an initiative, CoFI[1], to design *a Common Framework for Algebraic Specification and Development* was started [18]. The goal of CoFI is to get a common agreement in the algebraic specification community about basic concepts, and to provide a family of specification languages at different levels, a development methodology and tool support. The family of specification languages comprises of a central, common language, called CASL[2], various restrictions of CASL, and various extensions of CASL (e.g. with facilities for particular programming paradigms).

The definition of CASL and some of its sublanguages has been finished [8]. Moreover, a complete formal semantics of CASL [9] has been developed in parallel with design of the language and indeed, the development of the semantics has given important feedback to the language design.

[1] CoFI is pronounced like ‘coffee’.

[2] CASL is an acronym for *CoFI Algebraic (or* Axiomatic*) Specification Language* and is pronounced like ‘castle’.

Now that design and semantics of CASL have been finished, it is essential to have a good tool support. Tools will be essential for the goal of CoFI to get CASL accepted in academic communities (in the short run), and, in the long run, in industry. This holds even stronger since CASL is a language with a formal semantics: many people believe that such a language cannot or will not be used in practice: "The best semantics will not win." [13]

Since CASL was designed with the goal to subsume many previous frameworks, it has become a powerful and quite complex language. This complexity makes it harder to build tools covering the whole language.

In this work, we will show that it is possible to build tools for a complex language with strong semantics in a reasonable time. In order to achieve this, we have followed several guidelines:

- As much as possible, re-use existing tools, instead of building new ones.
- Build tools in such a way that tools developed at different sites can be integrated; thus, not every site has to develop all the tools.
- Make tools as generic as possible. After all, CASL only is the central language in a whole family of languages, and it would be tedious to have to re-implement the same things for each language separately.

All these guidelines are even more important in a non-commercial environment as the CoFI initiative is, where only very limited (wo)man-power is available, and therefore collaborative effort is essential. Moreover, an explicit goal within the design of CASL was to provide a common language in order to achieve a better interoperability of (already existing) tools.

We will discuss these guidelines, reporting how well they work in practice and which difficulties arise with them.

The paper is organized as follows:

Section 2 gives a brief overview over CASL and its semantics. Section 3 explains the general architecture of the Bremen HOL-CASL tool. In section 4, tool interoperability using a common interchange format is discussed. Section 5 describes the problems with parsing CASL's mixfix syntax. Section 6 recalls the encoding of CASL in higher-order logic from [17], while section 7 reports our practical experiences when using this encoding to create an interface from CASL to Isabelle/HOL. In section 8, some difficulties of encoding CASL structured specifications into Isabelle are discussed. Section 9 describes several user interfaces for HOL-CASL. Finally, section 10 contains the discussion how the guidelines work in practice, and directions for future work.

This work is based on [17], but considerably extends the work begun there.

2 CASL

CASL is a specification language that can be used for formal development and verification of software. It covers both the level of requirement specifications, which are close to informal requirements, and of design specifications, which are close to implemented programs. CASL provides constructs for writing

– basic specifications (declarations, definitions, axioms),
– structured specifications (which are built from smaller specifications in a modular way),
– architectural specifications (prescribing the architecture of an implementation), and
– specification libraries, distributed over the Internet.

Basic CASL specifications consist of declarations and axioms representing theories of a first-order logic in which predicates, total as well as partial functions, and subsorts are allowed. Predicate and function symbols may be overloaded [4]. Datatype declarations allow to shortly describe the usual datatypes occurring in programming languages.

Structured specifications allow to rename or hide parts of specifications, unite, extend and name specifications. Moreover, generic specifications and views allow to abstract from particular parts of a specification, which makes the specification reusable in different context.

Architectural specifications allow to talk about implementation units and their composition to an implementation of a larger specification (or, vice versa, the decomposition of an implementation task into smaller sub-tasks).

Structured and architectural specifications together with libraries will be also referred to as CASL-in-the-large, while basic specifications will be referred to as CASL-in-the-small.

The semantics of CASL follows a natural semantics style and has both rules for static semantics (which are implemented by a static semantic checker) and model semantics (which are implemented by theorem-proving tools).

spec LIST [**sort** $Elem$;] =
  free type $List[Elem] ::= nil \mid \_\_ :: \_\_(head :?\ Elem; tail :?\ List[Elem])$;
  %list $[\_\_], nil, \_\_ :: \_\_$
  op $\_\_ {+\!+} \_\_ : List[Elem] \times List[Elem] \to List[Elem]$;
  %prec $\_\_ :: \_\_ < \_\_ {+\!+} \_\_$
  vars $e : Elem$;
    $K, L : List[Elem]$
  • $\%[concat\_nil]\ nil {+\!+} L = L$
  • $\%[concat\_cons]\ (e :: K) {+\!+} L = e :: K {+\!+} L$
end

Fig. 1. Specification of lists over an arbitrary element sort in CASL

Consider the specification of lists over an arbitrary element sort in Fig. 1. The **free type** construct is a concise way to describe inductive datatypes. The semantic effect is the introduction of the corresponding constructor (here *nil* and $\_\_ :: \_\_$) and (partial) selector (here *head* and *tail*) functions, and of a number of axioms: a so-called sort generation constraint stating that the datatypes are inductively generated by the constructors and possibly by parameter sorts (here: the sort `Elem`), and first-order axioms expressing that the constructors are injective and have disjoint images and that the partial selectors are one-sided inverses of the corresponding constructors.

The annotation **%list** $[\_\_], nil, \_\_ :: \_\_$ allows to write lists in the form $[t_1, \ldots, t_n]$. This notation is not restricted to lists: with **%list**, one also can introduce abbreviating notations for sets, bags, etc.

3 Tool Architecture

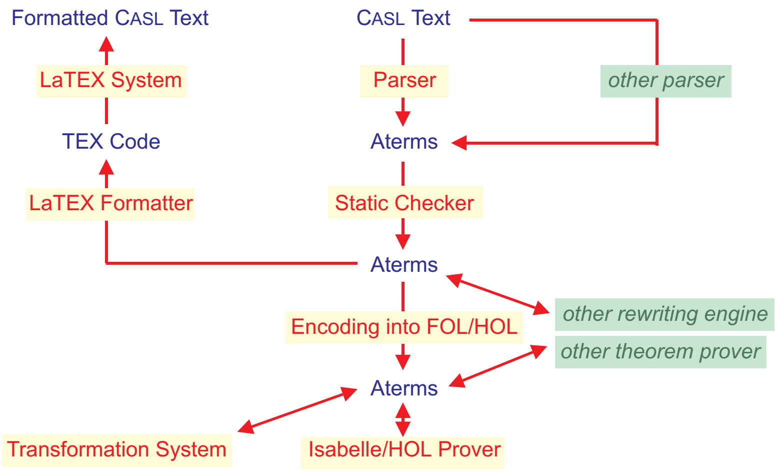


Fig. 2. Architecture of the HOL-CASL system

The Bremen HOL-CASL system consists of several parts, which are shown in Fig. 2. The *parser* checks the syntactic correctness of a specification (CASL Text) according to the CASL grammar and produces an abstract syntax tree (coded as ATerms). The *static checker* checks the static semantic correctness (according to the static semantics) and produces a global environment (also coded as ATerms) that associates specification names with specification-specific information such as the signature. The *LaTeX formatter* allows to pretty print CASL specifications (which are input in ASCII format), using the CASL LaTeX package from Peter Mosses [19]. For example, the specification in Fig. 1 has been generated from the ASCII input shown in Fig. 4.

Finally, the *encoding* is a bridge from CASL to first- or higher-order logic (FOL/HOL). It throws out subsorting and partiality by encoding it [17], and thus allows to re-use existing theorem proving tools and term rewriting engines for CASL. Typical applications of a *theorem prover* in the context of CASL are

– checking the model-semantic correctness of a specification (according to the model semantics) by discharging the proof obligations that have been generated during static semantic analysis,
– validate intended consequences, which can be added to a specification using an annotation. This allows a check for consistency with informal requirements,
– prove correctness of a development step (in a refinement).

4 Tool Interoperability

The are quite a number of existing specifications languages and tools for them. CASL was designed with the goal of providing a common language for better tool interoperability. This is reflected by having a common interchange format for CASL tools, the ATerm format [3]. ATerms are an easy-to-handle format with libraries in several languages (C, Java) available. The main reason for chosing ATerms was that the untyped term structures are very flexible, and their easy syntax makes it very easy to write parsers and printers for them (we needed to implement these in our implementation language, ML, which has been done very quickly).

Thus, ATerms are used as (untyped) low level tool format for data exchange between CASL tools. Based on this format, several (strongly typed) formats have been designed: the CasFix format [26] for abstract syntax trees, and a format for the global environment, containing the static semantic information.

A problem with ATerms is that the textual representation gets very large (the ATerm representation of the global environment for the CASL basic data types is about 10 MB). [3] have solved this problem by providing a compact binary format with full sharing of subterms. This format can deal efficiently even with Gigabyte-sized structures. However, parsers and printers for this format are more complex. Thus, we are using converters between the textual and the binary ATerm format written in C as a workaround, until an ML-based ATerm library dealing also with the binary format becomes available.

By providing conversions from and to ATerms at all intermediate points in the tool architecture, the Bremen HOL-CASL system can be used as a front-end or back-end in combination with other tools. Actually, it has been combined as a back-end with the Amsterdam CASL parser [27], and as a front-end with several theorem proving tools: ELAN [21], PVS [2] and Isabelle (see section 7). See also the CoFI Tools Group home page [10].

5 Parsing and Static Semantic Analysis

Apart from having a relatively complex grammar, CASL has several features that cause some difficulties for parsing and static analysis:

1. CASL's syntax allows user-defined mixfix syntax,
2. CASL allows mutually recursive subsort definitions, causing loops within a naive subsorting analysis, and
3. CASL allows overloading, and formulas which have a unique overload resolution up to semantical equivalence.

Concerning mixfix syntax, we separate parsing into two steps: The first pass of parsing produces an abstract syntax tree where formulas and terms (i.e. those parts of the specifications that may contain mixfix symbols) remain in their unparsed textual form.

Mixfix grouping analysis can be done only after a first phase of static semantic analysis has collected the operation and predicate symbols (among them the mixfix symbols). The CASL grammar is then extended dynamically according to the mixfix declarations, and formulas and terms are parsed with the generic Isabelle parser, which uses the well-known Cocke-Younger-Kasami algorithm for context-free recognition [11]. This grammar-parameterised algorithm has a complexity of $O(n^3)$, which is quite acceptable, since formulas and terms in CASL specifications are not that long. However, it turned out to be too slow to do the first pass of parsing with this approach. Therefore, we moved to ML-yacc for the first pass.

After having done the parsing of terms and formulas, those resulting parse trees are selected that are precedence correct with respect to the user-specified precedence relations. If more than one parse tree remains, the corresponding term or formula is ambiguous, and the possible disambiguations are output to the user. To obtain a concise output, not all pretty-printed forms of the parse trees are shown, but only the local places at which they actually differ.

The definition of precedence correctness follows the one of [1], generalized to CASL's pre-order based precedences ([1] uses number based precedences).

Concerning static semantic analysis, the treatment of subsorts and overload resolution needs a careful algorithmic design in order not to run into an exponential time trap. The details of this have already been worked out in [17].

6 Encoding CASL into HOL

In this section, we briefly recall the encoding from CASL into HOL from [17]:

At the level of CASL basic specifications, the encoding into higher-order logic proceeds in three steps:

1. The CASL logic, subsorted partial first-order logic with sort generation constraints (SubPCFOL), is translated to subsorted first-order logic with sort generation constraints (SubCFOL) by encoding partiality via error elements living in a supersort.
2. Subsorted first-order logic with sort generation constraints (SubCFOL) is translated to first-order logic with sort generation constraints (CFOL) by encoding subsorting via injections (actually, this is built-in into the CASL semantics [4]).

3. First-order logic with sort generation constraints (CFOL) is translated to higher-order logic (HOL) by expressing sort generation constraints via induction axioms.

These encodings are not only translations of syntax, but also have a model-theoretic counterpart[3], which provides an implicit soundness and completeness proof for the re-use of HOL-theorem provers for theorem proving in the CASL logic SubPCFOL. This is also known as the "borrowing" technique of Cerioli and Meseguer [5], which allows to borrow theorem provers across different logics.

7 The Interface to Isabelle/HOL

Using the encoding described in the previous section, we have built an interface from CASL to Isabelle/HOL. We have chosen Isabelle [20] because it has a very small core guaranteeing correctness. Furthermore, there is over ten years of experience with it (several mathematical textbooks have partially been verified with Isabelle). Last but not least, Isabelle is generic, i.e. it supports quite a number of logics, and it is possible to define your own logic within Isabelle. Despite the genericity of Isabelle, we have refrained from building the CASL logic directly into Isabelle – this would violate our guideline to re-use existing tools as much as possible: we would have to set up new proof rules, and instantiate the Isabelle simplifier (a rewriting engine) and tableau prover from scratch. Instead, we re-use the Isabelle logic HOL, for which already sophisticated support is available, with the help of the encoding described in section 6.

This encoding has a clear semantical basis due to the borrowing (most other encodings into Isabelle/HOL do not have an explicit model-theoretic counterpart). However, a good semantic basis does not imply that there are no practical problems:

First, the encoding of CASL in Isabelle/HOL as described in [17] produces too complex output. We had to fine-tune the output by suppressing superfluous parts (for example, trivial subsort injections), while retaining its mathematical correctness.

Another problem with borrowing is that the HOL-CASL user really works with the encoding of a CASL specification, and not with the CASL specification itself. In particular, goals and subgoals are displayed as HOL formulas, and the proof rules are of course the Isabelle/HOL proof rules. However, a typical user of the tool will probably be more familiar with CASL than with Isabelle/HOL. Therefore, we have decided to display goals and subgoals in a CASL-like syntax as much as possible. For example, an injection of a term t from a subsort $s1$ to a supersort $s2$ is displayed as $t : s2$, as in CASL, and not as $inj_{s1,s2}(t)$, as the encoding would yield. In this way, we get a CASL-like display syntax of Isabelle/HOL. Let us call this display syntax "CASLish Isabelle/HOL".

However, note that the CASLish Isabelle/HOL omits some information, e.g. the information that an injection $inj_{s1,s2}(t)$ starts from $s1$. In some practical example proofs, this turned out to be rather confusing (while in others, the longer

[3] Formally, they are institution representations in the sense of [16,24]

form $inj_{s1,s2}(t)$ is just tedious), and one would like to go back to the "pure Isabelle/HOL" view of the subgoals instead of using the "CASLish Isabelle/HOL". Therefore, we plan to let the user choose among several pretty printing "views" on his or her encoded CASL specification.

Another example for the mixture of CASL and Isabelle/HOL are Isabelle's two different kinds of free variables, which may occur in CASL formulas during a proof. Isabelle one one hand has object variables, which cannot be instantiated during a proof. They are used for proofs of universally quantified sentences. The other kind of variables are meta variables, which can be instantiated during a proof. They are used for proofs of existentially quantified sentences (cf. Prolog, narrowing). For example, when trying to prove

$$\exists\ x : Nat \ \bullet \ x + 9 = 12$$

by elimination of the existential quantifier, one gets

$$?x + 9 = 12$$

and then $?x$ is instantiated with 3 during the proof (while the goal $x + 9 = 12$ would not be provable, since $\forall\ x : Nat \ \bullet \ x + 9 = 12$ is false).

```
Level 0
((K ++ L) ++ M) = (K ++ (L ++ M))
 1. ((K ++ L) ++ M) = (K ++ (L ++ M))

Level 1
((K ++ L) ++ M) = (K ++ (L ++ M))
 1. !!x1 x2.
       ((x2 ++ L) ++ M) =
       (x2 ++ (L ++ M)) =>(((x1 :: x2) ++ L) ++ M) =
                          ((x1 :: x2) ++ (L ++ M))
 2. ((nil ++ L) ++ M) = (nil ++ (L ++ M))

Level 2
((K ++ L) ++ M) = (K ++ (L ++ M))
No subgoals!
```

Fig. 3. Proof of `forall K,L,M:List[Elem] . (K++L)++M=K++(L++M)`

Fig. 3 shows a proof of the associativity of the concatenation of lists, using the specification from Fig. 1. Level 0 shows the original goal. In the first proof step (level 1), the goal was resolved with the sort generation constraint for lists. The two subgoals are the inductive arguments for `__::__` and `nil`, respectively. In the second step, both subgoals can be proved feeding the axioms `concat_nil` and `concat_cons` into Isabelle's simplifier (a rewriting engine).

Another problem is that of input of goals. Goals are of course *input* in the CASL syntax (only during a proof, they get redisplayed in CASLish Isabelle/HOL syntax). One would like also to be able to input goals in Isabelle/HOL, for example when one needs to prove a lemma that is formulated in Isabelle/HOL. We solve this by providing Isabelle/HOL as a theory within our interface, and we parse goals that are input for this theory always with the Isabelle/HOL parser, and not with the CASL parser.

8 Encoding of CASL Structured Specifications

The encoding of structured specifications is almost orthogonal to that of basic specifications and therefore can be done in a generic, logic-independent way.

When encoding CASL structured specification into Isabelle, the problem arises that the structuring mechanism of CASL and Isabelle are rather different. In particular, Isabelle's mechanisms are considerably weaker: Extensions and unions of specifications are available in Isabelle (though the union is defined is a slightly different way), while for CASL's renamings, hidings, and generic specifications, nothing similar is available in Isabelle.

Currently, we solve this problem by just flattening structured specifications to basic specifications, that is, we literally carry out all the renamings, unions etc. Hidings can be treated by renaming the symbol which shall be hidden with a unique name that cannot be input by the user.

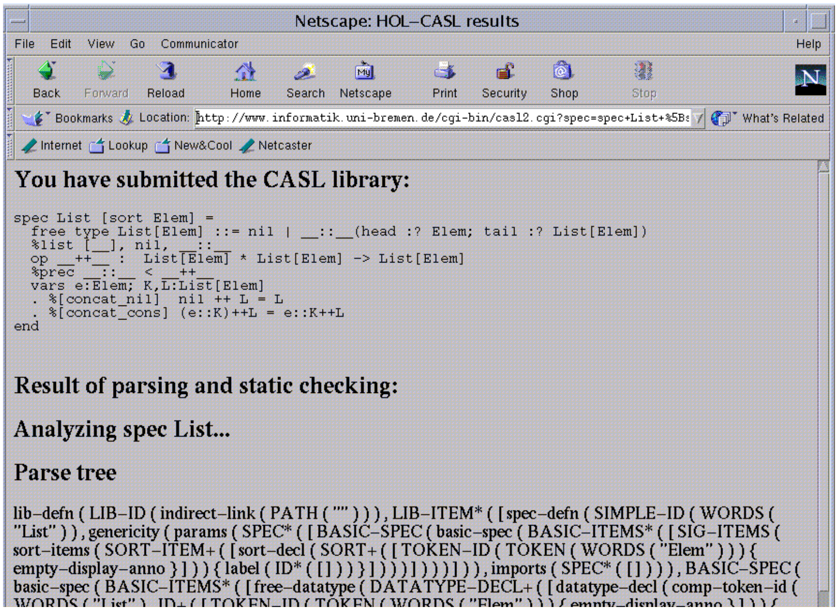
However, this is not very satisfactory, since flattening destroys the structural information of a specification and thus makes theorem proving in the specification harder. In some cases, the loss of structural information makes it practically infeasible to do proofs which are doable when the structuring is kept. Therefore, we have asked the Isabelle implementors to improve Isabelle's structuring mechanisms, and they have promised to do something in this direction.

In principle, an alternative way would be to use a deep encoding of CASL, which means to directly describe the semantics of CASL within higher-order logic. However, this would not be very nice, since theorem proving in a deep encoding is relatively far away from proving in the encoded logic. In contrast, we use a shallow encoding, where proving in the encoding comes close to proving in the encoded logic. The advantage of a deep encoding would be that one can prove meta-properties about the semantics of CASL, but in our view, this does not outweigh the disadvantages.

An exceptional case are CASL's free specifications. One can hardly expect to implement them in a logic-independent way, since they depend on an involved construction in the model categories of the logic. All that one can expect here is to simulate the semantics of free specifications in a particular logic within higher-order logic, along the lines of [23].

Encoding of architectural specifications is beyond the scope of this paper – it will be dealt with elsewhere.

As described in the previous section, libraries are an orthogonal matter. However, there is one important incompatibility between CASL and Isabelle at this



You have submitted the CASL library:

```
spec List [sort Elem] =
  free type List[Elem] ::= nil | __::__(head :? Elem; tail :? List[Elem])
  %list [__], nil, __::__
  op __++__ :  List[Elem] * List[Elem] -> List[Elem]
  %prec __::__  <  __++__
  vars e:Elem; K,L:List[Elem]
  . %[concat_nil]  nil ++ L = L
  . %[concat_cons] (e::K)++L = e::K++L
end
```

Result of parsing and static checking:

Analyzing spec List...

Parse tree

lib–defn (LIB–ID (indirect–link (PATH (""))) , LIB–ITEM* ([spec–defn (SIMPLE–ID (WORDS ("List")) , genericity (params (SPEC* ([BASIC–SPEC (basic–spec (BASIC–ITEMS* ([SIG–ITEMS (sort–items (SORT–ITEM+ ([sort–decl (SORT+ ([TOKEN–ID (TOKEN (WORDS ("Elem"))) { empty–display–anno }]))) { label (ID* ([ ])) }]))))]))))])) , imports (SPEC* ([ ]))) , BASIC–SPEC (basic–spec (BASIC–ITEMS* ([free–datatype (DATATYPE–DECL+ ([datatype–decl (comp–token–id (WORDS ("List") , ID+ ([TOKEN–ID (TOKEN (WORDS ("Elem"))) { empty–display–anno }])) {

Fig. 4. The web interface of the HOL-CASL system

point: CASL text files may contain libraries consisting of several specifications, while Isabelle text files always consist of exactly one Isabelle theory. We solve this problem by just splitting a CASL library into small files containing one specification each, and feeding these files into Isabelle. Or course, we also have to maintain the information associating a CASL library with the split files.

9 User Interface

We provide several user interfaces to the Bremen HOL-CASL system. Actually, it has turned out that for the first contact with our tool, the most important user interface is the web-based interface[4], where the user can just type in a specification, and parse it, perform the static analysis and/or conversion to LaTeX. Most users want to try out this easy-to-use interface before taking the effort to download the stand-alone version (even if the latter effort is very small). The web-interface has even been used as a front-end in a prototype translation to PVS [2] (although it is much more convenient to use the stand-alone version in this case).

[4] You can play around with it: `http://www.informatik.uni-bremen.de/cgi-bin/casl2.cgi`.

The small stand-alone version of our tool[5] provides the full functionality shown in Fig. 2, except the Isabelle theorem proving environment. It has been quite crucial to exclude Isabelle here, since Isabelle is quite large, and users who want to use the tool as a front-end or back-end do not want to download the whole Isabelle system. The stand-alone tool can be called as a Unix command, and the different entry points and phases of analysis and encodings of the tool (cf. Fig. 2) can be selected with optional flags. In particular, it is also possible to select the encoding into FOL/HOL without having to use Isabelle (this is useful when combining our tool with theorem provers for first- or higher-order logic). We also plan to make the different steps of the encoding (see section 6) separately available, so that one can choose to "encode out" just partiality and keep the subsorting (this will be useful, for example, in connection with Maude [6] which supports subsorting). The Unix interface works quite well when using the tool in combination with other tools, although we plan to provide a fully-fledged applications programmer interface (API) in the future.

The full stand-alone version of the tool[6] also provides the Isabelle theorem prover, and the generic graphical user interface IsaWin [15,14], which has been built on top of Isabelle. We have instantiated IsaWin with our HOL-CASL encoding of CASL into Isabelle/HOL. In Fig. 5, you can see a typical IsaWin window. The icons labelled with (Σ, E) are CASL specifications (more precisely, their encodings in HOL). Note that the theory HOL itself also is available at this level. The icon labelled with a tree is an open proof goal. By double-clicking on it, you can perform proof steps with this goal. This is done by dragging either already proven theorems (those icons marked with $\vdash A$) or simplifier sets (icons marked with $\{l \rightarrow r\}$) onto the goal. The effect is the resolution of the goal with the theorem thrown onto it, or the rewriting of the goal with the chosen simplifier set. After the proof of a goal is finished, it turns into a theorem. You can then use it in the proof of other theorems, or, if it has the form of a rewrite rule, add it to a simplifier set.

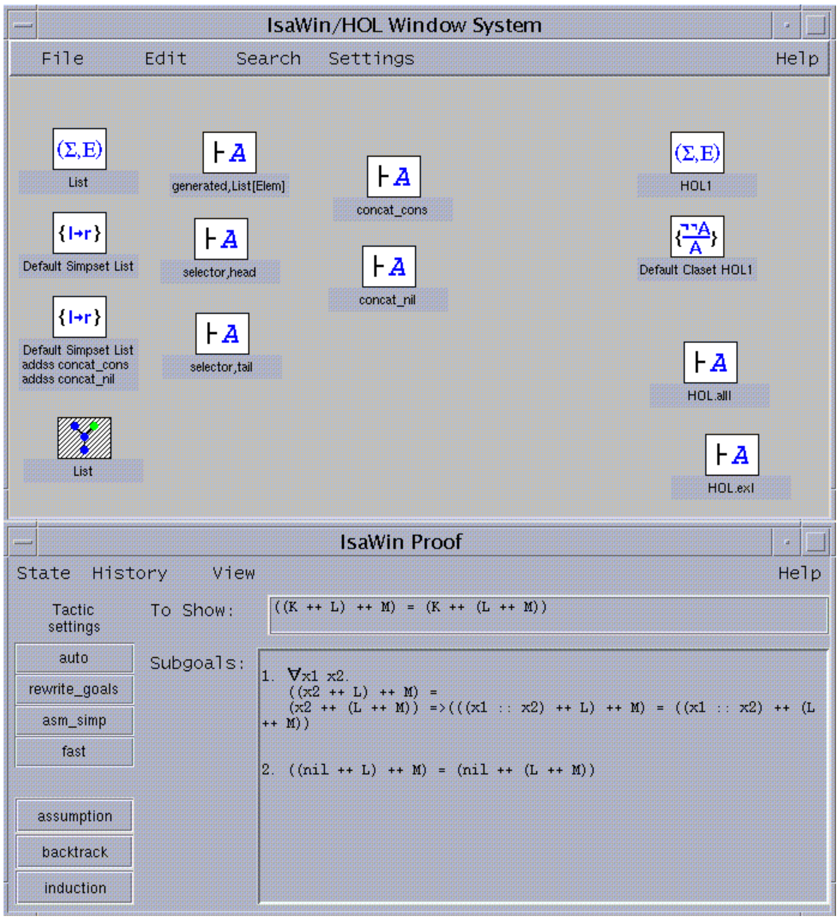
Actually, some users explicitly told us that they feared to have to install Isabelle to run our tool. However, even the full version including Isabelle and IsaWin is completely stand-alone (apart from the need to install Tcl/Tk, which has already been installed on many sites).

10 Conclusion and Future Work

We have shown that it is possible to write tools for a complex language with strong semantical bias (though it turns out to be a complex task). We could reduce the amount of work by re-using existing tools as much as possible. Moreover, by using a common tool interchange format, we have created a tool which can be used in connection with other tools as a front end or back end. Cur-

[5] Available at `http://www.informatik.uni-bremen.de/~cofi/CASL/parser/parser.html`.

[6] Available at `http://www.informatik.uni-bremen.de/~cofi/CASL/`.



```
((K ++ L) ++ M) = (K ++ (L ++ M))
```

```
1. ∀x1 x2.
   ((x2 ++ L) ++ M) =
   (x2 ++ (L ++ M)) =>(((x1 :: x2) ++ L) ++ M) = ((x1 :: x2) ++ (L
++ M))

2. ((nil ++ L) ++ M) = (nil ++ (L ++ M))
```

Fig. 5. The HOL-CASL instantiation of the IsaWin system

rently, our tool has been used in connection with two theorem provers (PVS and Isabelle) and one rewriting engine (ELAN).

We have followed three guidelines when implementing the HOL-CASL system. The first guideline was to re-use existing tools, rather than create a new tools. In practice, this has turned out to be very hard: Building an interface from CASL to an existing tool is quite a complex task, which not only deals with an input-output-transformation, but also has to take the interactive behaviour and the display of intermediate results into account.

Nevertheless, we think that it is worth the effort to re-use existing tools, since these tools have evolved and improved over time, and in a sense we borrow this maturity from other tools, which otherwise would only have been achieved through a long process of testing, use and maintenance. Of course, our bridges to other tools also have to become mature, but since the target of the bridging tools are already mature, the whole effort can start from a higher level.

Currently, we have re-used Isabelle/HOL for having a quick prototype of theorem proving environment for CASL, at the price to get a very HOLish CASL. In future work, we will develop derived rules and tactics for CASL (especially for the computation of normal forms w.r.t. the overloading axioms that state coincidence of overloaded functions on sub- and supersorts). With this, we will try to make the encoding to look more CASL-like by eliminating the need to work with HOL rules and instead provide a complete set of rules for CASL. Perhaps in a further step, we will even encode CASL directly in the generic Isabelle meta logic. Anyway, this step would probably have been too complicated in the first place, and working with Isabelle/HOL has the advantage of faster having a prototype.

Concerning the guideline of genericity, we have made the experience that the *use* of generic tools at some points can lead to inefficiencies: we had to replace the generic Isabelle parser by ML-yacc to obtain an efficient parsing. Yet, we have to use the generic parser for parsing user-defined mixfix syntax. Another experience was with the IsaWin system: it has been designed as a generic window-based interface to Isabelle, but when instantiating it to HOL-CASL, several changes to IsaWin were needed to make it actually useful. Nevertheless, the genericity was a great help in comparison to implementation from scratch.

Regarding genericity of our own tool, we have made the encoding of structured specifications independent of the underlying logic. One important future point will be to make also the static analysis of CASL structured and architectural specifications truly generic, i.e. also parameterized over a logic (this is possible because the semantics is already parameterized over a logic). This would allow to re-use the tool also for other logics than the logic underlying CASL (for example, higher-order CASL, reactive CASL, temporal logic, or just your own favourite logic).

Concerning interoperability, the use of ATerms helped a lot to interconnect our parser and static analysis with several theorem proving and rewriting tools at different other sites. Here, it was essential to use the very easy-to-handle textual ATerm representation to get quick prototypes of such interconnections, although for larger applications, the more complex binary format is needed.

Another use of the ATerm format will be the comparison of outputs of different tools for the same purposes that have been developed at different sites.

We hope that also tools developed by others will be integrated to work with our tools in the future. Currently, we have ATerm-based formats for parse trees and global static environments. For the integration of different theorem proving and rewriting tools, on would also need ATerm-based formats for proofs, proof states and possibly also transformations.

An even better integration can be achieved with the UniForM workbench [12], which also provides library management and access to a generic transformation application system [15,14] that will be instantiated to CASL.

Future work will turn our tool into a theorem proving environment that can be used for practical problems. On the way to this goal, we have to implement proof management, dealing with proof obligations, intended consequences and refinement. Moreover, special simplifiers and proof tactics for CASL will have to be developed an tested. A first case study will be the verification of proof obligations and intended consequences for the libraries of CASL basic datatypes [22].

Another direction of research will further exploit the possibility of the generic analysis of CASL-in-the-large. It is possible to extend CASL to a heterogeneous specification language, where one can combine specifications written in several different logics, see [25] for some first ideas. Tool support for such a language would extend the generic analysis of CASL-in-the-large with an analysis of structuring mechanisms for moving specifications between different logics.

Acknowledgements

I would like to thank Kolyang for an intensive cooperation lasting for several years. Christoph Lüth and Burkhart Wolff patiently helped with all kinds of questions about Isabelle and IsaWin. Pascal Schmidt did the implementation of the conversion from and to ATerms and of the LaTeX pretty printer. Markus Roggenbach wrote the first larger specifications in CASL, which were extensively used and tested with the tools. Bernd Krieg-Brückner provided a good working environment and discussions about the general directions of this work.

References

1. A. Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, May 1995. 98
2. D. Baillie. Proving theorems about CASL specifications. Talk at the 14th Workshop on Algebraic Development Techniques, Bonas, France, September 1999. 97, 102
3. M. Brand, H. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. Technical report, CWI, 1999. Accepted by SPE. 97
4. M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive subsorted partial logic in CASL. In M. Johnson, editor, *Algebraic methodology and software technology: 6th international conference, AMAST 97*, volume 1349 of *Lecture Notes in Computer Science*, pages 91–107. Springer-Verlag, 1997. 95, 98
5. M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173:311–347, 1997. 99
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag LNCS 1631. System Description. 103

7. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[7] and FTP[8]. 107
8. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [7], Oct. 1998. 93
9. CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (version 0.95), in [7], Mar. 1999. 93
10. CoFI Task Group on Tools. The CoFI-Tools group home page. http://www.loria.fr/∼hkirchne/CoFI/Tools/index.html. 97
11. J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, Reading, MA, 1979. 98
12. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. The UniForM Workbench, a universal development environment for formal methods. In *FM99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1186–1205. Springer-Verlag, 1999. 105
13. P. G. Larsen. VDM and proof rules for underdetermined functions. Talk at the IFIP WG 1.3 meeting, Bonas, France, September 1999. 94
14. C. Lüth, H. Tej, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program developkment and theorem proving. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99*, number 1577 in LNCS, pages 239–243. Springer Verlag, 1999. 103, 105
15. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167–189, Mar. 1999. 103, 105
16. J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989. 99
17. T. Mossakowski, Kolyang, and B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In F. Parisi Presicce, editor, *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, volume 1376 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 1998. 94, 96, 98, 99
18. P. D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997. Documents/Tentative/Mosses97TAPSOFT, in [7]. 93
19. P. D. Mosses. Formatting CASL specifications using LaTeX. Note C-2, in [7], June 1998. 96
20. L. C. Paulson. *Isabelle - A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag, 1994. 99
21. C. Ringeissen. Demonstration of ELAN for rewriting in CASL specifications. Talk at the 14th Workshop on Algebraic Development Techniques, Bonas, France, September 1999. 97
22. M. Roggenbach and T. Mossakowski. Basic datatypes in CASL. Note M-6, in [7], Mar. 1999. 106
23. P. Y. Schobbens. Second-order proof systems for algebraic specification languages. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 321–336, 1994. 101

[7] http://www.brics.dk/Projects/CoFI
[8] ftp://ftp.brics.dk/Projects/CoFI

24. A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer Verlag, 1996. 99
25. A. Tarlecki. Towards heterogeneous specifications. In D. Gabbay and M. van Rijke, editors, *Frontiers of Combining Systems, 2nd International Workshop*. Research Studies Press, 1999. To appear. 106
26. M. van den Brand. CasFix – mapping from the concrete CASL to the abstract syntax in ATerms format. http://adam.wins.uva.nl/∼markvdb/cofi/casl.html, 1998. 97
27. M. G. J. van den Brand and J. Scheerder. Development of parsing tools for CASL using generic language technology. Talk at the 14th Workshop on Algebraic Development Techniques, Bonas, France, September 1999. 97