

# CCC – The CASL Consistency Checker

Christoph Lüth<sup>1</sup>, Markus Roggenbach<sup>2</sup>, and Lutz Schröder<sup>1</sup>

<sup>1</sup> Department of Mathematics and Computer Science, Universität Bremen, Germany

<sup>2</sup> Department of Computer Science, University of Wales Swansea, United Kingdom

**Abstract.** We introduce the CASL Consistency Checker (CCC), a tool that supports consistency proofs in the algebraic specification language CASL. CCC is a faithful implementation of a previously described consistency calculus. Its system architecture combines flexibility with correctness ensured by encapsulation in a type system. CCC offers tactics, tactical combinators, forward and backward proof, and a number of specialised static checkers, as well as a connection to the CASL proof tool HOL-CASL to discharge proof obligations. We demonstrate the viability of CCC by an extended example taken from the CASL standard library of basic datatypes.

## 1 Introduction

Consistency of specifications is an important issue: validating a specification by proving intended consequences (sanity or conformance checking) is meaningless without a consistency proof – *ex falso quodlibet* –, and implementing a specification is impossible in the presence of inconsistencies. Some formal development paradigms and specification languages handle this problem by excluding inconsistent specifications. In contrast, algebraic specification languages such as CASL [4, 13] allow inconsistent specifications. This allows the developer to concentrate on the desired properties of the system during the requirements engineering phase, then validate their consistency in a separate, later step, before finally proceeding to implement the specification.

This paper describes a prototype of the CASL Consistency Checker (CCC), a tool that supports consistency proofs for CASL specifications. It is a faithful implementation of the previously introduced calculus for consistency proofs of CASL specifications [15]. CCC is part of wider effort to provide tool support for CASL, comprising the CASL tool set CATS [12] which includes a parser, static analysis, and an encoding into higher-order logic, which is used to embed CASL into Isabelle/HOL [14], thus providing proof support for CASL specifications (HOL-CASL).

The material is structured as follows: in Sect. 2, we review the basic concepts of the consistency calculus [15]. We then describe the system architecture in Sect. 3 and show the CCC at work with an extended example in Sect. 4.

## 2 The Consistency Calculus

The specification language CASL [4, 13] constitutes a standard in algebraic specification. Its features include total and partial functions, predicates, subsorted overloading, sort generation constraints, and structured and architectural specifications. A method for proving consistency of CASL specifications has been introduced in [15]; we briefly recall the main features of the calculus.

The consistency calculus comprises three parts, concerned with specification equivalence, conservativity of extensions, and definitionality of extensions, respectively. The core of the method is the conservativity calculus; consistency of a specification is encoded as conservativity over the empty specification. The implementation extends the calculus of [15] by well-formedness assertions, so that well-formedness also of unparseable specifications (namely, specifications that contain specification variables) can be guaranteed.

**The Extension Calculus.** This calculus handles extension judgements of the form  $Sp_1 \preceq Sp_2$  which state that one has a signature inclusion which is a specification morphism  $Sp_1 \rightarrow Sp_2$ . Equivalence of specifications  $Sp_1 \simeq Sp_2$  is defined as mutual extension. These notions of extension and equivalence are meant to be used only for minor syntactical adjustments; in particular, the extension calculus is not intended as a means to establish so-called views, which serve to describe general specification morphisms in CASL. Typical rules of the calculus state that  $(Sp_1 \textbf{ then } Sp_2)$  extends  $Sp_1$ , that the union of specifications is idempotent, commutative, and associative, and that  $(Sp_1 \textbf{ then } Sp_2)$  is equivalent to  $(Sp_1 \textbf{ and } Sp_2)$ , provided the latter is well-formed (this is an example where a well-formedness assertion is needed).

**The Definitionality Calculus.** An extension  $Sp_1 \preceq Sp_2$  is called *definitional* if each model of  $Sp_1$  extends *uniquely* to a model of  $Sp_2$ ; in CASL, this is expressed by the semantic annotation `%def`. In particular, definitionality implies conservativity (see below). A definitionality assertion is written

$$def(Sp_1)(Sp_2).$$

The definitionality calculus plays an auxiliary role, since the main concern of the method is conservativity. It presently covers definition by abbreviation and primitive recursion; further extensions such as well-founded recursion are obvious, but require more elaborate tool support.

**The Conservativity Calculus.** The notion of conservativity denoted by the CASL annotation `%cons` is that of model extensivity: an extension  $Sp_1 \preceq Sp_2$  is *conservative* if each model  $M$  of  $Sp_1$  extends to a model of  $Sp_2$ ; this is written

$$cons(Sp_1)(Sp_2).$$

The consistency assertion  $c(Sp)$  abbreviates  $cons(\{\})(Sp)$ , where  $\{\}$  denotes the empty specification. Since the empty specification has a unique model,  $c(Sp)$  indeed states that  $Sp$  is consistent. The conservativity rules divide into three major groups:

- Basic language-independent rules, typical examples being a rule that states that conservative extensions compose and a rule which allows deducing conservativity from definitionality.
- Logic-independent rules that propagate conservativity along the various CASL structuring constructs. E.g., unions of specifications are treated by the rule

$$\begin{array}{c}
 \begin{array}{l}
 Sp_i \text{ defines the signature } \Sigma_i, i = 1, 2 \\
 \Sigma_1 \cup \Sigma_2 \text{ is amalgamable}
 \end{array}
 \quad
 \begin{array}{l}
 Sp \preceq Sp_1, Sp \preceq Sp_2 \\
 Sp \text{ defines } \Sigma_1 \cap \Sigma_2 \\
 cons(Sp)(Sp_1)
 \end{array} \\
 \text{(union)} \quad \frac{\quad}{cons(Sp_2)(Sp_1 \text{ and } Sp_2)}
 \end{array}$$

Approximative algorithms for checking amalgability are already implemented in CATS. This is a typical case where static side conditions are relegated to further tools integrated into CCC.

- Logic-specific rules that guarantee conservativity for certain syntactic patterns such as data types or positive Horn extensions. A simple example is

$$\text{(free)} \quad \frac{newSort(DD_1 \dots DD_n)(Sp) \quad Sp \text{ then types } DD_1; \dots; DD_n \text{ has a closed term for each new sort}}{cons(Sp)(Sp \text{ then free types } DD_1; \dots; DD_n)}$$

where the  $DD_i$  are datatype declarations and the assertion  $newSort(DD_1 \dots DD_n)(Sp)$  states that the sorts declared in  $DD_1 \dots DD_n$  are not already in  $Sp$  – another example of a proof obligation that is discharged by a static checker.

The strategy for conservativity proofs is roughly as follows: the goal is split into parts using the logic-independent parts of the conservativity calculus, occasionally using the extension calculus for certain sideward steps; at the level of basic specifications, conservativity is then established by the definitionality calculus and the logic-specific rules of the conservativity calculus. This may involve the use of built-in static checkers, and, at eventually pinpointed hot spots, actual theorem proving.

### 3 System Architecture

For a tool such as a consistency checker or theorem prover, *correctness* is critical: if the tool asserts that a specification is consistent, we need to be sure that this follows from the consistency calculus, not from a bug in the implementation. On the other hand, *flexibility* is important as well: users should be as unconstrained as possible in the way which they conduct their consistency proofs.

CCC's design follows the so-called LCF design [8], where a rich logic (such as higher-order logic) is implemented by a small *logical core* of basic axioms and inference rules. In this design, the logical core implements an abstract datatype of theorems, with logically correct inference rules as operations. Other theorems can only be derived by applying these operations, i.e. by correct inferences; thus

the correctness of the whole system is reduced to the correctness of the logical core. The logic encoded within the logical core is called the *meta-logic*, whereas the logic being modelled by the rules is the *object logic*.

Figure 1 shows the system architecture in three layers: innermost, we have the logical core, surrounded by the extended object logic which supplements the meta-logic with specialised proof procedures. The outermost layer is given by auxiliary proof infrastructure.

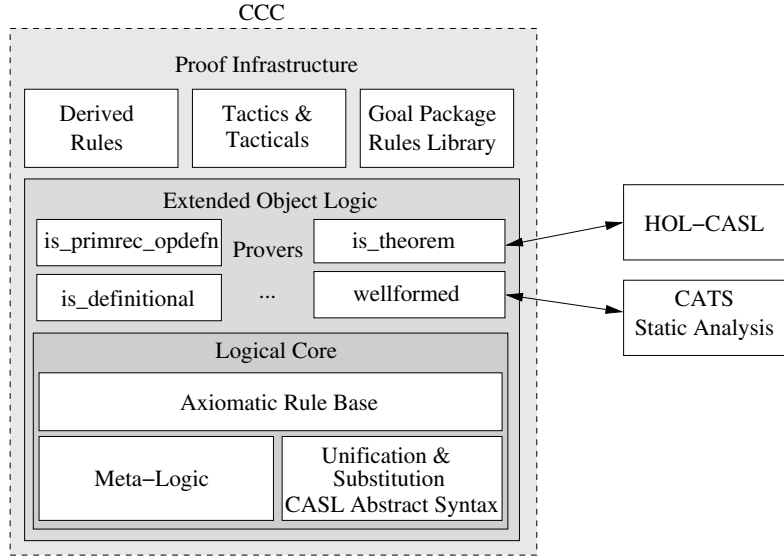


Fig. 1. CCC System Architecture

### 3.1 The Logical Core

The logical core of CCC implements the meta-logic, the axioms of the object logic, and the axiomatic rule base.

The *meta-logic* is a weak fragment of conjunctive logic. It formalises *rules* as we have seen in Sect. 2 above, and ways in which to manipulate them. A rule allows us to deduce a proposition, the *conclusion*, from a list of propositions, the *premises*. All deductions live in the context of a particular (global) environment which maps *names* to specifications; hence, all rules are parameterised by an environment. We write such a rule as  $\Gamma \vdash P_1, \dots, P_n \rightarrow Q$ , where  $\Gamma$  is an environment,  $P_1$  to  $P_n$  are the premises, and  $Q$  is the conclusion. Figure 2 shows the rules of the meta-logic (where  $P\sigma$  is the application of a substitution  $\sigma$  to a proposition  $P$ , and  $mgu$  is the most general unifier of two propositions). This formulation of the meta-logic allows proofs by both forward and backward reso-

lution. Forward resolution is application of the meta-rule `composei` and allows us to derive a new rule from two old ones. Backward resolution allows goal-directed proof (see Sect. 3.3 below).

$$\begin{array}{c}
 \overline{\Gamma \vdash P \rightarrow P} \text{ assume} \\
 \\
 \frac{\Gamma \vdash P_1, \dots, P_n \rightarrow Q \quad \Delta \vdash R_1, \dots, R_m \rightarrow S \quad \sigma = \text{mgu}(Q, R_i) \quad \begin{array}{l} 1 \leq i \leq m \\ \Gamma \subseteq \Delta \end{array}}{\Delta \vdash R_1\sigma, \dots, R_{i-1}\sigma, P_1\sigma, \dots, P_n\sigma, R_{i+1}\sigma, \dots, R_m\sigma \rightarrow S\sigma} \text{ compose}_i \\
 \\
 \frac{\Gamma \vdash P_1, \dots, P_n \rightarrow Q \quad \begin{array}{l} i \neq j, 1 \leq i, j \leq n, \\ P_i = P_j \end{array}}{\Gamma \vdash P_1, \dots, P_{j-1}, P_{j+1}, \dots, P_n \rightarrow Q} \text{ contract}_{i,j} \\
 \\
 \frac{\Gamma \vdash P_1, \dots, P_n \rightarrow Q}{\Gamma \vdash P_1\sigma, \dots, P_n\sigma \rightarrow Q\sigma} \text{ specialise}
 \end{array}$$

**Fig. 2.** Inference Rules of the CCC meta-logic

The *object logic* implements the judgements of the conservativity calculus. They are modelled by a datatype `prop`, with CATS used to model the abstract syntax of CASL (in particular, `AS.L_SPEC` is the type of specifications):

```

datatype prop = consistent_SPEC of AS.L_SPEC
              | conservative of AS.L_SPEC * AS.L_SPEC
              | definitional of AS.L_SPEC * AS.L_SPEC
              | implicational of AS.L_SPEC * AS.L_SPEC
              | ext of AS.L_SPEC * AS.L_SPEC
              | equiv of AS.L_SPEC * AS.L_SPEC
              | provable of pprop

```

The datatype `pprop` is explained in detail below.

The third component of the logical core is the *rule base*. This is a collection of rules the correctness of which has to be proved outside of the system by appealing to the CASL semantics, as opposed to all other rules, which are derived from these rules using the meta-rules; in other words, the rule base contains all rules of the consistency calculus of [15] except the ones explicitly stated as derived.

### 3.2 Provable Propositions and Provers

The *extended object logic* adds decision procedures, called *provers*, to the logical core. Provers apply to specific proof obligations called *provable propositions*

represented by the datatype `pprop`. There are about thirty kinds of provable propositions; an excerpt of the datatype `pprop` showing three typical cases is this:

```
datatype pprop = well_formed of AS.L_SPEC
                | is_just_signature of AS.L_SPEC
                | is_theorem of AS.L_SPEC * AS.FORMULA list | ...
```

The first type of proof obligations states that a particular specification is well-formed; this can be discharged by calling CATS’ static analysis. The second states that a specification is merely a signature, and can be discharged by a straightforward recursive function which descends the syntax tree of the specification and returns false as soon as it finds something which does not belong into a signature (such as axioms or free datatypes). The third says that a list of formulae is provable from the given specification, and requires interactive theorem proving using HOL-CASL. A typical rule that has a provable proposition as a premise is the conservativity rule for subtype definitions,

$$\text{(sub)} \quad \frac{\text{newSort}(s)(Sp) \quad \text{implies}(Sp)(Sp \text{ then axiom } \exists v : t \bullet F)}{\text{cons}(Sp)(Sp \text{ then sort } s = \{v : t \bullet F\})}$$

Here, the provable proposition  $\text{newSort}(s)(Sp)$  states that the sort  $s$  is not already declared in  $Sp$ , a fact that is easily checked statically.

By distinguishing propositions (`prop`) and proof obligations (`pprop`), we restrict the potential harmful effects of wrongly implemented provers. For example, it is impossible to write a prover which returns `consistent_SPEC(Sp)` for every specification  $Sp$ . Note that provers are supplied when the system is built, never at run-time by the user.

### 3.3 Proof Infrastructure

The proof infrastructure contains further modules which facilitate interactive or semi-automatic proof. *Derived rules* are those of the rules from the calculus [15] which can be derived from the rule base. The *tactics package* allows us to write advanced proof procedures. A *tactic* is a function on rules. The rules of the meta-logic give us elementary tactics, which together with combinators such as case distinction or sequential composition can be composed to more sophisticated tactics such as one which handles all definitional extensions. The *rule library* stores and retrieves previously shown results, and the *goal package* allows backwards proof, starting from a stated goal and reducing it to the list of empty premises by tactics application.

Users interact with the system using the SML command line interface, or more comfortably using an instance of the Proof General interface [1]. The latter combines SML’s flexibility and expressional power with script management and a comfortable interactive environment.

CCC consists of about 7500 lines of SML code (excluding CATS and HOL-CASL). It runs under SML of New Jersey, but should be easily portable to other SML implementations. Source code and binary builds can be downloaded from the CCC web site [5].

## 4 Extended Example

To demonstrate the CCC's capabilities, we will show the consistency of the specifications NAT of natural numbers (see Fig. 3) and CHAR of the datatype of ASCII characters (see Fig. 4), taken from the CASL standard library of Basic Datatypes [13]<sup>1</sup>. The simple structure of these specifications allows a detailed discussion of their respective CCC proof scripts. However, the proofs involve non-trivial consistency arguments. Furthermore, in the case of the specification CHAR a complete consistency proof is not feasible without tool support due to the length of the specification which involves more than 1000 axioms.

```
spec Nat = free type Bool ::= TT | FF
  then free type Nat ::= 0 | suc(pre:? Nat)
  then op __<=__: Nat * Nat -> Bool
  ...
  then op __ * __ : Nat * Nat -> Nat;
  forall m,n : Nat
  . 0 * m = 0
  . suc(n) * m = (n * m) + m
  then op 1: Nat = suc (0); ...; op 9: Nat = suc (8);
  op __ @@ __ (m:Nat;n:Nat): Nat = (m * suc(9)) + n
end
```

**Fig. 3.** The specification NAT

```
spec Char = Nat
  then sort Byte = { n: Nat . (n <= 255) = TT }
  then free type Char ::= chr(ord: Byte)
  then op '\000' : Char = chr(0 as Byte);
  ...
  then op NL:Char = LF;
  then op '\n' : Char = NL;
end
```

**Fig. 4.** The specification CHAR

### 4.1 Consistency Proof of NAT

Figure 5 shows the CCC proof script. We start by loading the library containing NAT (`load_lib "Numbers"`), stating our goal, and unfolding the specification

<sup>1</sup> For the purposes of this paper, the specification text has been slightly modified to make the consistency proof more readable.

```

1  load_lib "Numbers"; ccc "Nat";                (* start the proof *)
2  ap (compose' Struct.name1);                  (* unfold the spec *)
3  ap (Repeat(OpDefns));                        (* deal with Op defns *)
4  ap (Repeat prim_rec_defns);                  (* deal with prim rec defs *)
5  ap (prove_free_type "0" 1);                  (* deal with free type Nat,
6  "0" as witness for non-empty carrier set *)
7  ap (prove 2 Prover.p_holcasl_auto);
8  ap (compose Struct.add_empty 1); (* add empty spec as start point *)
9  ap (prove 1 Prover.p_well_formed);
10 ap (prove_free_type "TT" 1); (* deal with free type Bool:
11 "TT" as witness for non-empty carrier set *)
12 ap (prove 2 Prover.p_holcasl_auto);
13 ap (compose' Basic.triv_consistency); (* empty spec is consistent *)
14 ap (prove 1 Prover.p_is_just_signature);
15 qeccc "Nat";                                (* store the result *)

```

**Fig. 5.** The CCC proof script for the specification NAT

(lines 1–2). The general idea of consistency proofs in CCC is to reduce the overall goal to simpler goals by working backwards through the specification text, reducing it to structures simple enough to show their consistency directly.

In our example, the first step is to show that the operation definition

$$\text{op } \_ \text{ @@ } \_ \text{ (m:Nat;n:Nat): Nat} = (\text{m} * \text{suc}(9)) + \text{n}$$

is definitional. If this is the case, the whole specification NAT is consistent if its specification text without the last line is consistent. This type of argument (the tactic `OpDefns`, line 3) can be repeated for all the digit definitions from `op 1: Nat = suc (0)` to `op 9: Nat = suc (8)`. Here, we can use the tactical combinator `Repeat`, which applies its argument until it fails. Applying the composed tactic reduces our goal to consistency of this smaller specification:

```

free type Bool ::= TT | FF
then ...
then op _ * _ : Nat * Nat -> Nat;
forall m,n : Nat
. 0 * m = 0
. suc(n) * m = (n * m) + m

```

Here, multiplication on natural numbers is a new operation whose axioms are primitive recursive. This is verified by the tactic `prim_rec_defns` (line 4). Again, this type of argument can be repeated, as also `++` and `<=` are defined by primitive recursion. Hence, we have reduced the specification to be shown consistent to

```

free type Bool ::= TT | FF
then free type Nat ::= 0 | suc(pre:? Nat)

```

```

1  load_lib "SimpleDatatypes"; ccc "Char";          (* start the proof *)
2  ap (compose' Struct.name1);                    (* unfold the spec *)
3  ap (Repeat(OpDefns));                          (* deal with the Op Defns *)
4  ...
5  ap (prove 3 Prover.p_new_sorts_closed_terms_dd);
6                                          (* deal with free type *)
7  ap (specialize_with (("t", 0), "chr(0 as Byte)" 3);
8  ap (prove 3 Prover.p_closed_term_for_sort);
9  holcasl 3; ...; caslqed();
10
11 ap (compose SpecialExt.sub 2);                (*deal with subsort definition*)
12 ap (prove 2 Prover.p_new_sort);
13 ap (compose Imp.theorem_prover_basic 2 );
14 holcasl 2; by (rtac exI 1);                  (* proof in HOL-CASL *)
15 by (rtac leq_def1_Nat 1); caslqed ();
16
17 ap (compose' Struct.named);                  (* use the result c(Nat) *)
18 ap (compose' (get "Nat"));
19 qeccc "c_Char";                              (* store the proof *)

```

Fig. 6. The CCC proof script for the specification CHAR

Next, we deal with the definition of the natural numbers as a free type. The tactic `prove_free_type` (line 5) takes 0 as a witness that there exists a defined term of type `Nat`. In the next step, the definedness of 0 is verified by simple theorem proving in HOL-CASL. These arguments reduce the specification text relevant for consistency to

```
free type Bool ::= TT | FF
```

Now we add the empty specification (lines 8–9), as justified by our equivalence rules:

```
{ } then free type Bool ::= TT | FF
```

This allows us to apply again the tactic `prove_free_type` (line 10), this time with `TT` as a witness. After discharging the proof obligation that `TT` is defined (line 12), it remains to prove that the empty specification is consistent (line 13). This is verified by the prover `Prover.p_is_just_signature`, which checks that the empty specification does not contain any axioms. Finally, we store our consistency result under the name `Nat` for later re-use (line 15).

## 4.2 Consistency Proof of CHAR

For this example, see the proof script in Figure 6, we need to load the library `SimpleDatatypes` (line 1), which imports the specification `NAT` the consistency of which we have shown in the previous section. After stating our proof goal (line

1) and unfolding the specification (line 2), the first actual proof steps consist of showing that all the operation definitions `op ' \ n' : Char = NL` till `' \000' : Char = chr(0 as Byte)` are definitional (remember that we are working backwards). For this purpose we repeat again the tactic `OpDefns` (line 3). This reduces our goal to show the consistency of this smaller specification:

```
Nat then sort Byte = { n: Nat . (n <= 255) = TT }
  then free type Char ::= chr(ord: Byte)
```

To deal with `free type Char ...`, we have to show that the sort `Char` is new and non-empty. The prover `Prover.p_new_sorts_closed_terms_dd` checks the first condition (line 5) and generates a proof obligation, where the user has to provide a closed term as a witness that the carrier of the sort `Char` is non-empty; here, we choose `"chr(0 as Byte)"` (line 6) and can then discharge the proof obligation (line 7)<sup>2</sup>. This leaves us with the proof obligation that `chr(0 as Byte)` is actually defined, which we discharge with a small HOL-CASL proof (line 8; the details of the proof are elided here).

Similarly, to deal with the subsort definition `Byte`, we need to check that sort `Byte` is new (`Prover.p_new_sort`, line 11) and its carrier is non-empty. To this end, we need to show that there exists an element in the sort `Nat` which is less or equal to 255 (rule `Imp.theorem_prover_basic`, line 12), which requires more theorem proving in HOL-CASL (lines 13–14).

We finish the proof by recalling the consistency of `Nat` using the above established result (line 16–17). This is possible, because the specification `Nat` has been imported and hence is part of the global environment in which we prove the consistency of `Char`. Finally, the established theorem `"c_Char"` is stored with the command `qeccc` (line 18).

## 5 Conclusions and Future Work

CCC is a tool to support consistency proofs for specifications written in the standard algebraic specification language CASL. The calculus implemented by CCC supports a proof method where large specifications are split into parts along their explicit specification structure; trivial consistency issues are discharged along the way, leading to the real hot spots of the specification that possibly require actual theorem proving. As presented here, the tool should be seen as a prototype and research vehicle, which we can use to study how to conduct consistency proofs for large, realistic CASL specifications using our calculus; it is certainly not a ready-to-use industrial strength tool yet.

The design of CCC focuses on two main issues: firstly, by basing the design on a small and encapsulated logical core, *correctness* of the tool reduces to the correctness of this core, i.e. essentially correctness of the calculus in [15].

---

<sup>2</sup> These are the steps which we combined to the tactic `prove_free_type` used in the previous section in the consistency proof of `NAT`.

Secondly, tactics, tactical combinators, forward and backward proof give users the *flexibility* to conduct consistency proofs in a comfortable and extensible way, and to design powerful proof strategies. This allows us to gradually develop effective and efficient proof tactics for realistic specifications.

The use of CCC has been illustrated by means of an extended example. Further experiments include specifications from the libraries NUMBERS, RELATIONSANDORDERS, ALGEBRA\_I from the CASL standard library of Basic Datatypes [13] as well as consistency checks of datatypes evolving in an industrial case study of specifying an electronic payment system [7]. While logically simple, these examples provide enough material both in terms of structure and size to show not only that the tool is able to deal with substantial specifications, but that its use indeed represents added value.

*Related work:* We can distinguish between approaches which avoid inconsistency by construction, and approaches which show consistency by showing satisfiability. The first approach comprises model-based specification formalisms such as Z [16] and VDM [10], where a model of the system is constructed rather than an axiomatic description; systems based on conservative extension such Isabelle [14], where specifications are built by conservatively extending consistent ones; and systems based on constructive type theory such as Coq [3] or Alfa/Agda [9, 6]. Following the second approach, there is a huge body of work on the satisfiability (and hence consistency) of first-order formulae, which is complimentary to our work; in our terminology, such automatic tools are provers which can be used to prove the consistency of a set of axioms, and we aim to integrate these tools into our system in the future. However, the contribution of our work is to provide a framework in which to conduct consistency proofs for large, structured specifications.

*Future work:* We will focus on designing more powerful tactics by testing the tool with more examples selected from a wide range of case studies including specifications found in [4, 13, 2, 7]. These examples will be more involved at the level of basic specifications (i.e. in terms of logic rather than in terms of structuring). On the other hand, more decision procedures will be provided in order to increase the degree of automation. Obvious candidates include decision procedures already implemented in the CASL tool set (for example concerning the search for witnesses of non-emptiness of types), as well as existing automatic consistency checkers or SAT solvers such as Chaff [11].

## Acknowledgements

The authors would like to thank Janosch Neuweiler and Tobias Thiel for their help in implementing CCC, Erwin R. Catesbeiana for asking the right questions, Till Mossakowski for consultations on HOL-CASL and CATS, and David Aspinall for helping to set up the Proof General interface.

## References

1. D. Aspinall, *Proof General: A generic tool for proof development*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 1785, Springer, 2000, pp. 38–42.
2. H. Baumeister and D. Bert, *Algebraic specification in CASL*, Software Specification Methods: An Overview Using a Case Study (M. Frappier and H. Habrias, eds.), Springer, 2000.
3. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004.
4. M. Bidoit and P. D. Mosses, *CASL User Manual*, LNCS, vol. 2900, Springer, 2004.
5. *The CCC homepage*, <http://www.informatik.uni-bremen.de/cofi/ccc>.
6. C. Coquand, *Agda homepage*, <http://www.cs.chalmers.se/~catarina/agda>.
7. A. Gimblett, M. Roggenbach, and H. Schlingloff, *Towards a formal specification of an electronic payment system in CSP-CASL*, Recent Trends in Algebraic Development Techniques (WADT 204) (José Luiz Fiadeiro, Peter Mosses, and Fernando Orejas, eds.), LNCS, Springer, To appear.
8. M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: a Mechanised Logic of Computation*, LNCS, vol. 78, Springer, 1979.
9. T. Hallgren, *Alfa homepage*, <http://www.cs.chalmers.se/~hallgren/Alfa>.
10. C. B. Jones, *Systematic Software Development using VDM*, Prentice Hall, 1990.
11. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an efficient SAT solver*, Design Automation, ACM, 2001, pp. 530– 535.
12. T. Mossakowski, *CASL - from semantics to tools*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 1785, Springer, 2000, pp. 93–108.
13. P. D. Mosses (ed.), *CASL Reference Manual*, LNCS, vol. 2960, Springer, 2004.
14. T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS, vol. 2283, Springer, 2002.
15. M. Roggenbach and L. Schröder, *Towards trustworthy specifications I: Consistency checks*, Recent Trends in Algebraic Development Techniques (WADT 201), LNCS, vol. 2267, Springer, 2002, pp. 305–327.
16. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1992, 2nd edition.