Towards Trustworthy Specifications I: Consistency Checks

Markus Roggenbach and Lutz Schröder

BISS, Department of Computer Science, Bremen University

Abstract. As the first of two methodological devices aimed at increasing the trust in the 'correctness' of a specification, we develop a calculus for proving consistency of CASL specifications. It turns out to be possible to delegate large parts of the proof load to syntactical criteria by structuring consistency proofs along the given specification structure, so that only in rather few remaining focus points, actual theorem proving is required. The practical usability of the resulting calculus is demonstrated by extensive examples taken from the CASL library of basic data types.

Introduction

The verification of *programs* is a well-established topic in computer science; here, correctness of programs is usually defined w.r.t. requirements given in a more or less formal language, possibly an algebraic specification language such as the language CASL (*Common Algebraic Specification Language*) [7, **?**] designed by COFI, the international *Common Framework Initiative for Algebraic Specification and Development* [6]. However, these *requirement specifications* themselves may very well contain errors in the sense that they fail to have exactly the intended class of models; early detection of such errors will substantially reduce the cost of software development.

Since the notion of an 'intended' class of models is not equipped with a formal meaning, the correctness of a requirement specification in this sense can only be 'verified' in an approximative process. This process splits into two parts, corresponding to the two required inclusions between the actual and the intended model class.

In order to increase the trust in the claim that the given specification does not admit unwanted models, one can attempt to prove certain intended consequences from the given axioms. Thus, it is shown that at least those models which fail to have these intended properties are excluded from the actual model class. This program is systematically pursued in the forthcoming [20]. (The opposite strategy — constructing counterexamples for intended properties in order to detect faulty specifications — is developed, e.g., in [1] and in [18].) Conversely, of course, one has to make sure that none of the intended models fails to satisfy the specification. The first step in this direction is to show that the specification has at least one model, i.e. that it is (semantically) consistent. (Note that for CASL specifications, semantical consistency and syntactical consistency, i.e. absence

of contradictions, do not coincide, since CASL sort generation constraints and CASL free specifications are higher order concepts). It is important to note that the proof of intended consequences as discussed above is meaningless without a consistency proof — ex falso quodlibet.

Here, we develop a set of rules for checking consistency of CASL specifications in a systematic way. As these rules work along the actual specification text, the need to construct (and prove) actual models of specifications is avoided as far as possible. The given set of rules is certainly far from complete. However, we illustrate its applicability by demonstrating how it can be used to establish the consistency of a large part of the CASL Basic Datatypes [19]. In order to be able to deal with this example and other realistic specifications, we discuss the calculus for full CASL here rather than restrict to a possibly more digestible sublanguage.

The consistency rules are designed in such a way that they can be automated to a large extent: most of them can be statically checked, i.e. they make use of the static semantics only. Just in a few cases, e.g. in rules that assume the correctness of views, actual theorem proving is required to discharge proof obligations that arise from the model semantics.

Besides using certain syntactical criteria, the consistency rules rely heavily on the CASL structuring mechanisms and their semantic annotations. Consequently, consistency proofs follow the structure of the given specification. A simple example for this type of argument is the exploitation of specification morphisms that arise e.g. from instantiations or extensions for transporting consistency. In this way, our rules highlight the (usually few) 'hot spots' of a specification, while the (lengthy) 'trivial' parts of the consistency argument are discharged automatically.

The structuring of proofs along the structure of specifications is a well-established concept. Proof systems of this kind for statements expressed within the given logic are elaborated e.g. in [14]. The *development graph* [2] provides automatic proof support for this type of reasoning, as well as for structured 'meta-reasoning' (although such terminology is deemed unsuitable for minors [11]) about specification refinement; the latter point is also addressed in [4, 10]. The most central metapredicate used below is conservativity of extensions; reasoning about this predicate within the development graph is discussed in [17]. To the best of our knowledge, however, the question how to structure consistency proofs along the structure of specifications has not been addressed before.

1 CASL

The specification language CASL has been designed by CoFI, the international *Common Framework Initiative for Algebraic Specification and Development* [6]. Following [**?**], we present a short overview of those features of CASL which are relevant here. For the full definition of the language, we refer to [7].

Roughly speaking, a CASL *basic specification* consists of a *signature* made up of sorts, operations, and predicates (declared by means of the keywords **sort**, **op**,

and **pred**, respectively, optionally equipped with a defining term or formula), and axioms referring to the signature items (keywords **axiom**, **forall**). Operations can be partial or total. Furthermore, one may declare a *subsort relation* on the sort symbols. Axioms are written in first-order logic. Going one step beyond first order logic, CASL also features sort generation constraints for datatypes (keywords **generated**, **free type**).

A *model* of such a specification is an algebra which interprets the sorts as (non empty) sets and the operations and predicates as (partial) functions and subsets, respectively, in such a way that the given axioms are satisfied. The subsorting relation is reflected by injective coercion functions between the sets interpreting the involved sorts (*not* by subset inclusion); for a discussion of the difficulties arising from this subtlety and their solution see [21].

Moreover, CASL provides ways of building complex (*structured*) specifications out of simpler ones (the simplest ones being basic specifications) by means of various *specification-building operations*. These include translation, hiding, union, and both free and loose forms of extension.

Translations of declared symbols to new symbols are specified by giving lists of 'maplets' of the form $old \mapsto new$ (keyword **with**).

Reducing a specification means removing symbols from its signature and the corresponding items from its models. CASL provides two ways of specifying a reduction: by listing the symbols to be *hidden* (keyword **hide**), or by listing those to be left visible, i.e., *revealed* (keyword **reveal**). CASL also facilitates the hiding of auxiliary symbols by allowing the local scope of their declarations to be indicated (keyword **local**).

The signature of a *union* of two specifications is the union of their signatures. Given models over the component signatures, the unique model over the union signature that extends each of these models is called their *amalgamation*; a pair of models is called *compatible* if their amalgamation exists. Clearly, not all pairs of models over component signatures amalgamate: an obvious necessary condition is that the models coincide on the common symbols (including subsort embeddings) of the component signatures. The models of a union (keyword **and**) are all amalgamations of the models of the component specifications.

Extensions (keyword **then**) may specify new symbols or merely require further properties of old ones. Extensions can be classified by their effect on the model class specified. For instance, an extension is called *conservative* when no models are lost: every model of the specification being extended is a reduct of some model of the extended specification. CASL provides annotations %**implies**, %**def**, and %**cons** to denote that the model class is not changed, that each model of the specification can be uniqueley extended to a model of the extended specification, or that the extension is conservative, resp. It is important to note that these annotations have no effect on the semantics of a specification: a specifier may use them to express his intentions, tools may use them to generate proof obligations. In the consistency proofs of Section 3, we use these annotations as a guideline for selecting appropriate rules.

Structured specifications may be *named*, and a named specification may be *generic* in the sense that it declares *parameters* that need to be *instantiated* when the specification is (re)used. Instantiation is a matter of providing an appropriate *argument specification* together with a *fitting morphism* from the parameter to the argument specification. A generic specification may also declare *imports* (keyword **given**) which behave like immediately instantiated parameters; the formal parameters (as well as the argument specifications in any instantiation) are regarded as extensions of the imports.

To allow reuse of fitting 'views', specification morphisms (from parameters to arguments) may themselves be named (keyword **view**).

Specifications may be declared to be *closed* (keyword **closed**), which means that they behave as though there were no previously declared signature elements (this becomes relevant as soon as there are translations in the scope of a **closed** construct). Instantiations of generic specifications are implicitly closed.

The simplest case of a *free* specification (keyword **free**) is the one where the specification constrained to be interpreted freely is closed. The signature of the specification is unchanged, but its model class is restricted to the initial models. More generally, a free specification may be a free extension, e.g.:

sort *Elem* **then**
free
{ type $Set ::= \{\} \mid \{\_\_\}(Elem) \mid \_\_ \cup \_\_(Set; Set)$
op $\_\_ \cup \_\_ : Set \times Set \to Set, \mathbf{assoc}, \mathbf{comm}, \mathbf{idem}, \mathbf{unit}\ \{\}$ **}**

Many structured specifications can be flattened, i.e. transformed into basic specifications (although for general structured specifications, flattening is precluded by the presence of free specifications and hiding operations; cf. [4]). However, the strategy pursued here consists to a large extent in exploiting these structuring operations in order to obtain correspondingly structured proofs; therefore, flattening will not play any rôle.

2 The Calculus

The first point that needs to be stressed is that the calculus developed below is really not about consistency at all, but rather about *conservativity* of extensions. Intuitively speaking, an extension is conservative if it does not 'specify away' any models, i.e. if each model of the original specification can be enlarged to a model of the extended specification [22]. This can be formalized as follows: an extension $\sigma : Sp_1 \hookrightarrow Sp_2$ induces a model reduction

$$\mathbf{Mod}(\sigma) : \mathbf{Mod}(Sp_2) \to \mathbf{Mod}(Sp_1),$$

where $\mathbf{Mod}(Sp)$ may for the purposes of this paper be thought of as denoting the *class* of models of a specification Sp (cf. [5, 7, 8] for details). σ is *conservative* if $\mathbf{Mod}(\sigma)$ is surjective, and *definitional* if $\mathbf{Mod}(\sigma)$ is bijective. For both these properties, CASL offers *semantic annotations*: conservative and definitional extensions may be indicated by the annotations %**cons** and %**def**, respectively.

Note that definitionality of an extension implies that it does not declare any new sorts. There is a third annotation, %**implies**, which applies to definitional extensions that do not affect the signature.

Now it is trivial to observe that a specification is *consistent* (in the sense that its model class is non-empty) iff it conservatively extends the empty specification **{}**. Thus, it does indeed suffice to provide a calculus for conservativity (which would necessarily have formed a part of a consistency calculus anyway). Conservativity is in itself an important notion in many contexts; for a recent application, see e.g. [17]. Moreover, we sketch the beginnings of a definitionality calculus, which, in this context, serves primarily the purpose of improving the readability of conservativity proofs.

The judgements of the conservativity calculus are of the form $cons(Sp_1)(Sp_2)$, where $Sp_1 \hookrightarrow Sp_2$ is an extension (in a somewhat generalized sense to be made precise below). Similarly, definitionality is expressed by the predicate $def(Sp_1)(Sp_2)$. Moreover, we use a predicate $implies(Sp_1)(Sp_2)$ which corresponds to the annotation %**implies** in CASL, i.e. Sp_1 and Sp_2 have the same signature and class of models. This predicate is not supported by a calculus; instead, its verification will assumed to be discharged by a suitable theorem prover. Finally, we do introduce a consistency predicate $c(Sp)$; however, this is just an abbreviation for $cons(\mathbf{\{\}})(Sp)$. Only a few rules refer specifically to the consistency predicate.

The calculus has been minimized as far as possible in order to keep it manageable and understandable. Its actual application, in particular by a tool, will require a catalogue of derived rules in order to avoid overly clumsy proofs; some examples of such rules are discussed at the end of this section.

The rules are organized as follows: Figure 1 contains a simple extension calculus for CASL specifications. The conservativity calculus proper is then subdivided into general conservativity rules (Figure 2), rules for special extensions (Figure 3), rules for structuring constructs (Figure 4), and definitionality rules (Figure 5). Correctness proofs for the rules have been generally omitted.

In the following we use without further explanation the symbol system introduced in the CASL Language Summary [7], where e.g. a specification is denoted by Sp, FM is a fitting morphism, BI is a basic item, etc. *All rules are subject to the silent premise that the specifications they are applied to are well-formed.*

Extension Calculus In order to keep the number of rules in the conservativity calculus as small as possible, we begin by introducing an *extension calculus* for specifications (Figure 1). This calculus is purely auxiliary and does not have any claims to completeness (nor is it meant as a step towards a calculus for specification refinement); however, it does allow a rather more elegant formulation of many of the conservativity rules. The calculus concerns judgements of the form $Sp_1 \preceq Sp_2$, which are taken to mean that $Sp_1 \hookrightarrow Sp_2$ is an extension. Here, the word *extension* is to be understood as meaning that the signature defined by Sp_1 is a subsignature of that defined by Sp_2, and that the associated signature inclusion is in fact a specification morphism $Sp_1 \to Sp_2$. At the level of CASL, this can be rephrased as correctness of the view

view V: Sp_1 **to** Sp_2 **end**.

Most of the extension calculus is really about *equivalence* of CASL specifications. Equivalence of two specifications Sp_1 and Sp_2, denoted $Sp_1 \simeq Sp_2$, is defined as mutual extension; i.e.,

$$Sp_1 \simeq Sp_2 : \Longleftrightarrow \ (Sp_1 \preceq Sp_2 \text{ and } Sp_2 \preceq Sp_1).$$

Of course, $Sp_1 \simeq Sp_2$ is equivalent to $implies(Sp_1)(Sp_2)$. However, the two notations serve different methodological purposes: while $Sp_1 \simeq Sp_2$ is used to transform specifications, $implies(Sp_1)(Sp_2)$ denotes a proof obligation to be discharged by a suitable theorem prover. Again, it should be stressed that the notion $\simeq$ is subsidiary to the conservativity calculus and by no means aimed at normal forms of specifications and the like.

The relation $\preceq$ is reflexive and transitive; moreover, it is compatible with the structuring operations. This implies that the same holds for $\simeq$ (which is, moreover, trivially symmetric).

The union operator **and** is commutative and associative. It is also idempotent in the sense that a union of a specification and an extension of that specification is essentially the same as the extension. There are rules for 'unfolding' instantiations of (parametrized) specifications according to the circumscription in the summary. Since we are really interested in keeping the structure of specifications here, these rules have been included more for the sake of their obviousness than in order to be routinely applied, with the exception of the simplest case, which concerns instantiations of unparametrized named specifications.

The remaining rules of Figure 1 are largely self-explanatory, such as idempotency of the **closed**-construct, equivalence of extension and union under well-formedness of the latter, or introductions of extra **then**s. Of course, certain basic equivalences may be derived; e.g., by rules (inst-rp) and (cl), $SN[FA_1] \dots [FA_n] \simeq$ **closed** $SN[FA_1] \dots [FA_n]$.

General Rules Some basic rules of the conservativity calculus are shown in Figure 2. Besides the obvious rules on trivial extensions and composition of extensions and on the relation between the metapredicates *cons, def,* and *implies,* there is a weakening rule (wk), as well as rules (rp1), (rp2) that allow replacing specifications by equivalent ones. Of course, the premises concerning the relations $\preceq$ and $\simeq$ in the latter three rules are meant to be discharged by means of the extension calculus of Figure 1. As discussed above, the main purpose of these rules is the simplification of the calculus; they are intended for 'conservative' use, i.e. mostly for minor syntactical adjustments rather than, say, for wholesale flattening and the like.

Rules for Special Extensions These rules (Figure 3) provide mechanisms for dealing with features of basic specifications such as axioms or data type definitions; since we are not concerned with actual theorem proving here, their scope is necessarily somewhat limited. Some of the rules listed here are discussed in a similar context in [13, 16]. Most of them make use of a predicate *newSorts* which

$$\textbf{(refl)}\ \frac{}{Sp \preceq Sp} \qquad \textbf{(trans)}\ \frac{Sp_1 \preceq Sp_2 \quad Sp_2 \preceq Sp_3}{Sp_1 \preceq Sp_3}$$

$$\textbf{(cong)}\ \frac{Sp_1 \preceq Sp_1'}{\{Sp_1\ \textbf{then}\ Sp_2\} \preceq \{Sp_1'\ \textbf{then}\ Sp_2\}\ \text{etc.}} \qquad \textbf{(ext)}\ \frac{}{Sp_1 \preceq \{Sp_1\ \textbf{then}\ Sp_2\}}$$

$$\textbf{(hdx)}\ \frac{}{Sp\ \textbf{hide}\ SL \preceq Sp} \qquad \textbf{(rvx)}\ \frac{}{Sp\ \textbf{reveal}\ SL \preceq Sp}$$

$$\textbf{(rveq)}\ \frac{SL\ \text{lists the symbols renamed by}\ SM}{Sp\ \textbf{reveal}\ SM \simeq Sp\ \textbf{reveal}\ SL\ \textbf{with}\ SM}$$

$$\textbf{(union-c)}\ \frac{}{\{Sp_1\ \textbf{and}\ Sp_2\} \simeq \{Sp_2\ \textbf{and}\ Sp_1\}}$$

$$\textbf{(union-a)}\ \frac{}{\{\{Sp_1\ \textbf{and}\ Sp_2\}\ \textbf{and}\ Sp_3\} \simeq \{Sp_1\ \textbf{and}\ \{Sp_2\ \textbf{and}\ Sp_3\}\}}$$

$$\textbf{(union-i)}\ \frac{Sp_1 \preceq Sp_2}{Sp_2 \simeq \{Sp_1\ \textbf{and}\ Sp_2\}} \qquad \textbf{(inst-eq1)}\ \frac{SN = Sp}{SN \simeq \textbf{closed}\{Sp\}}$$

$$\textbf{(inst-eq2)}\ \frac{\begin{array}{c}SN[Sp_1]\dots[Sp_n] = Sp\\ Sp' = \{\{Sp_1\ \textbf{and}\dots\textbf{and}\ Sp_n\}\ \textbf{then}\ Sp\}\\ FA_i : Sp_i \to Sp_i',\ i = 1,\dots,n\\ FM\ \text{is the extension of the}\ FA_i\ \text{to}\ Sp'\end{array}}{SN[FA_1]\dots[FA_n] \simeq \textbf{closed}\{\{Sp'\ \textbf{with}\ FM\}\ \textbf{and}\ Sp_1'\ \textbf{and}\dots\textbf{and}\ Sp_n'\}}$$

$$\textbf{(inst-eq3)}\ \frac{\begin{array}{c}SN[Sp_1]\dots[Sp_n]\ \textbf{given}\ Sp_1'',\dots,Sp_m'' = Sp\\ Sp' = \{\{Sp_1''\ \textbf{and}\dots\textbf{and}\ Sp_m''\}\ \textbf{then}\ \{Sp_1\ \textbf{and}\dots\textbf{and}\ Sp_n\}\ \textbf{then}\ Sp\}\\ FA_i : \{\{Sp_1''\ \textbf{and}\dots\textbf{and}\ Sp_n''\}\ \textbf{then}\ Sp_i\} \to \{\{Sp_1''\ \textbf{and}\dots\textbf{and}\ Sp_n''\}\ \textbf{then}\ Sp_i'\},\ i = 1,\dots,n\\ FM\ \text{is the extension of the}\ FA_i\ \text{to}\ Sp'\end{array}}{SN[FA_1]\dots[FA_n] \simeq \textbf{closed}\{\{Sp'\ \textbf{with}\ FM\}\ \textbf{and}\ Sp_1'\ \textbf{and}\dots\textbf{and}\ Sp_n'\}}$$

$$\textbf{(cl)}\ \frac{}{\textbf{closed}\ \{Sp\} \simeq Sp}$$

$$\textbf{(double)}\ \frac{Sp\ \text{contains}\ BI}{Sp \simeq \{Sp\ \textbf{then}\ BI\}}$$

$$\textbf{(then-and)}\ \frac{Sp_1\ \textbf{and}\ Sp_2\ \text{is well-formed}}{\{Sp_1\ \textbf{then}\ Sp_2\} \simeq \{Sp_1\ \textbf{and}\ Sp_2\}}$$

$$\textbf{(sep1)}\ \frac{}{\{BI_1\dots BI_n\} \simeq \{BI_1\dots BI_k\ \textbf{then}\ BI_{k+1}\dots BI_n\}}$$

$$\textbf{(sep2)}\ \frac{}{Sp \simeq \{\}\ \textbf{then}\ Sp}$$

$$\textbf{(perm)}\ \frac{\begin{array}{c}\{BI_1\dots BI_k\ BI_{k+1}\dots BI_n\}\ \text{is well-formed}\\ \{BI_1\dots BI_{k+1}\ BI_k\dots BI_n\}\ \text{is well-formed}\end{array}}{\{BI_1\dots BI_k\ BI_{k+1}\dots BI_n\} \simeq \{BI_1\dots BI_{k+1}\ BI_k\dots BI_n\}}$$

$$\textbf{(wkfree)}\ \frac{}{\textbf{generated}\ \{SI_1\dots SI_n\} \preceq \textbf{free}\ \{SI_1\dots SI_n\}}$$

Fig. 1. The extension calculus

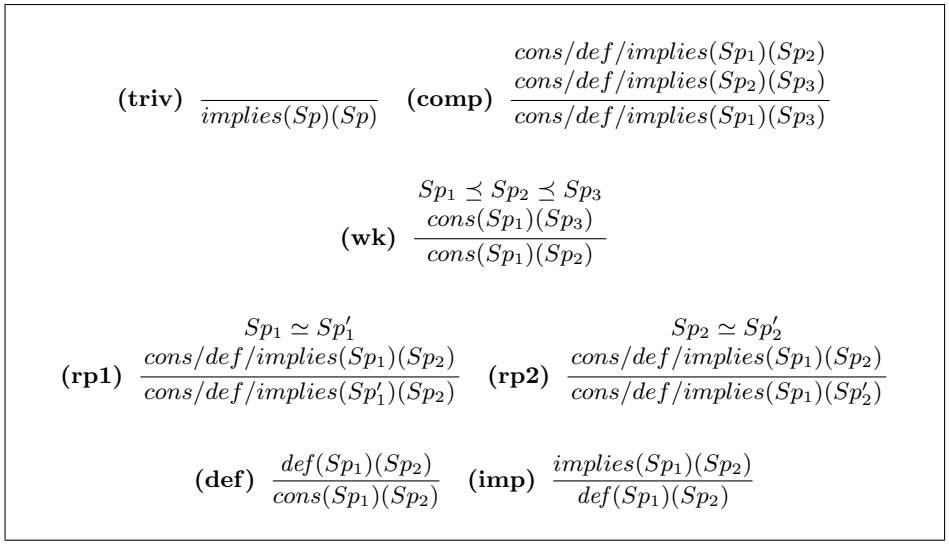


Fig. 2. General conservativity rules

expresses the (easily checkable) fact that a sort s or the sorts declared in signature items $SI_1, \ldots, SI_n$ or datatype declarations $DD_1, \ldots, DD_n$, respectively, are not contained in the signature of Sp. If a new sort is declared to be (freely) generated, it has to be checked whether the declared signature provides a closed term (i.e. a term without free variables) of this sort. This complication is due to the fact that the CASL semantics requires non-empty carriers.

Rule (horn) states that extensions in *positive* Horn logic are conservative if they have no effect on the previously declared sorts. The metapredicate $horn(Sp)$ is true for a basic specification iff all its axioms (including the implicit ones) are positive Horn clauses, possibly after performing skolemization, e.g. in the case of the existence axiom for inverses in a group; cf. Section 3.1. (Note that the axiom of choice is explicitly assumed in the CASL semantics.) Subsort declarations and **type** definitions (without sort generation constraints) can be coded by positive Horn clauses and hence are regarded as such. The *horn* predicate is recursively extended to structured specifications in the obvious way, except that hiding must be excluded here; i.e. if Sp is a specification in positive horn logic, then Sp **hide** SM need not be equivalent to a specification in positive Horn logic. (In the case of instantiations of parametrized specifications, the extension of the horn predicate involves additional proof obligations concerning well-formedness of the instantiation, i.e. correctness of the fitting morphism, in the same way as in rule (inst-eq3) of Figure 1.) Note that the phrase 'Conclusions in Sp_2 concern new predicates or equality in new sorts over Sp_1' refers also to implicitly generated axioms arising, e.g., from subsorting and overloading.

Further, rather obvious rules (rules (sub), (free), and (gtd1)) concern the introduction of subsorts, free datatypes, and (unrestricted) generated types, re-

spectively. The somewhat surprising satisfiability constraint expressed by the premise *implies*(...)(...) in rule (sub) is necessary due to the fact that the CASL semantics requires non-empty carriers; of course, discharging this constraint will in general require some form of theorem proving (even if the formula in question is in positive Horn form!). The case of generated types with a specified equivalence relation is somewhat more complicated. The rules provided here for this purpose (rules (gtd2) and (gtd3)) cover two syntactical patterns (the adherence to which is meant to be mechanically checkable): definition of equality via an observer or recursive definition of equality (where, for the time being, recursion is restricted to primitive recursion over the term structure). An observer is understood to be a function or a predicate which is defined by recursion over the term structure of the relevant data types. Observers may contain additional parameters of types other than the ones introduced in the datatype definition. A typical example of an observer is the elementhood predicate for finite sets, c.f. Section 3.2; using this observer, equality of sets is defined by the usual extensionality axiom. In both cases, the phrase 'defines equality' comes with a proof obligation, namely that the resulting relation on terms is indeed a congruence on the associated term algebra. If the 'equality' is defined by an observer, the equivalence axioms come for free, but the compatibility with the constructors remains as a proof obligation. Note that we have refrained from formulating explicit rules for the **generated types** construct, which, as in [7], is regarded as a special case of the **generated** construct.

Rules for Structuring Constructs As discussed above, consistency proofs profit from being designed following the specification structure. This requires rules that allow breaking down the structuring constructs of CASL. Rules of this type are listed in Figure 4; some of them are related to rules presented in [17].

The rules on translation, hiding, revealing, and local specifications (rules (tr), (hd1), (hd), (rv1), (rv2), and (local)) hardly require explanation. Rule (free) captures the fact that specifications in positive Horn form have initial models, provided there is a closed term for each sort. Rule (view) uses a 'correctness predicate' for CASL views; discharging this premise will in general require actual theorem proving. The rules on named specifications just state that a named specification is consistent if its 'unnamed version' is consistent.

The last three rules of Figure 4 are somewhat more involved. All of them require certain diagrams of *signatures* to be amalgamable; this is to be understood as follows: a commutative square

$$\begin{array}{ccc} \Sigma_1 & \longrightarrow & \Sigma_2 \\ \downarrow & & \downarrow \\ \Sigma_3 & \longrightarrow & \Sigma_4 \end{array}$$

of signature morphisms is called *amalgamable* if it is mapped to a pullback (of model classes or categories) under the model functor. Roughly speaking, this means that for each pair (M, N), where M and N are models of Σ_2 and Σ_3,

$$\textbf{(horn)}\ \frac{\begin{array}{c}horn(Sp_2)\\ \text{Conclusions in } Sp_2 \text{ concern new predicates or equality in new sorts over } Sp_1\end{array}}{cons(Sp_1)(Sp_1\ \textbf{then}\ Sp_2)}$$

$$\textbf{(sub)}\ \frac{\begin{array}{c}newSorts(s)(Sp)\\ implies(Sp)(Sp\ \textbf{then axiom}\ \exists\ v : t \bullet F)\end{array}}{cons(Sp)(Sp\ \textbf{then sort}\ s = \{v : t \bullet F\})}$$

$$\textbf{(gtd1)}\ \frac{\begin{array}{c}newSorts(SI_1 \ldots SI_n)(Sp)\\ Sp\ \textbf{then}\ SI_1 \ldots SI_n \text{ has a closed term for each new sort}\end{array}}{cons(Sp)(Sp\ \textbf{then generated}\ \{SI_1 \ldots SI_n\})}$$

$$\textbf{(gtd2)}\ \frac{\begin{array}{c}newSorts(SI_1 \ldots SI_n)(Sp)\\ Sp_1\ \textbf{then}\ SI_1 \ldots SI_n \text{ has a closed term for each new sort}\\ Sp_2 \text{ recursively defines an observer } f\\ Sp_3 \text{ defines equality on new sorts by } f\end{array}}{cons(Sp_1)(Sp_1\ \textbf{then generated}\ \{SI_1 \ldots SI_n\}\ \textbf{then}\ Sp_2\ \textbf{then}\ Sp_3)}$$

$$\textbf{(gtd3)}\ \frac{\begin{array}{c}newSorts(SI_1 \ldots SI_n)(Sp)\\ Sp_1\ \textbf{then}\ SI_1 \ldots SI_n \text{ has a closed term for each new sort}\\ Sp_2 \text{ recursively defines equality on new sorts}\end{array}}{cons(Sp_1)(Sp_1\ \textbf{then generated}\ \{SI_1 \ldots SI_n\}\ \textbf{then}\ Sp_2)}$$

$$\textbf{(free)}\ \frac{\begin{array}{c}newSorts(DD_1 \ldots DD_n)(Sp)\\ Sp\ \textbf{then types}\ DD_1; \ldots; DD_n \text{ has a closed term for each new sort}\end{array}}{cons(Sp)(Sp\ \textbf{then free types}\ DD_1; \ldots; DD_n)}$$

Fig. 3. Rules for special extensions

respectively, such that the respective reductions of M and N to Σ_1 agree, there exists a unique model of Σ_4 that extends M and N ('compatible models can be amalgamated'). This requirement is *approximated* by the condition that the above diagram forms a pushout in the signature category. However, there are certain additional difficulties that arise from subsorting; see [21] for a detailed discussion. In fact, whether or not a diagram as above is amalgamable is, in general, undecidable; however, practically relevant cases are covered by a polynomial algorithm which is currently implemented in the CASL tool set [15].

The relevant point here is that conservativity of the morphism $\Sigma_1 \to \Sigma_2$ in the above diagram implies conservativity of the morphism $\Sigma_3 \to \Sigma_4$, provided that the diagram is amalgamable (cf. [9]). A typical example is given by instantiations of parametrized specifications: a parametrized specification $SN[Sp_1] = Sp$ (for the sake of simplicity without imports and with only one parameter) defines, at the level of signatures, a signature extension $\Sigma_F \to \Sigma_B$, where Σ_F is the signature of the formal parameter Sp_1 and Σ_B that of the body. An instantiation of this parametrized specification requires a fitting morphism $FA : Sp_1 \to Sp_1'$; this fitting morphism is, in particular, a signature morphism $\Sigma_F \to \Sigma_A$, where Σ_A is the signature of the actual parameter Sp_1'. The result of the instantiation $SN[FA]$ then has signature Σ_R, where

$$\begin{array}{ccc} \Sigma_F & \longrightarrow & \Sigma_B \\ \downarrow & & \downarrow \\ \Sigma_A & \longrightarrow & \Sigma_R \end{array}$$

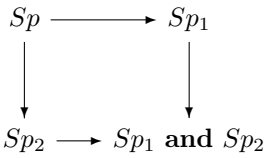
is a pushout in the signature category. In many typical situations, the body turns out to be a conservative extension of Sp_1. Thus, amalgamability of the above diagram allows us to deduce that $SN[FA]$ is conservative over Sp_1'; this is the essence of rules (inst2) and (inst3). (Of course, rule (inst3) may be regarded as subsuming rule (inst2); the latter has been included for sake of its better readability.) Note that this type of reasoning may substantially decrease the proof burden, since the non-trivial task of verifying amalgamability can in practically relevant cases be delegated to the above-mentioned algorithm.

There is an additional twist to the — otherwise similar — rule (union). At the level of signatures, a union Sp_1 **and** Sp_2 corresponds to a diagram

$$\begin{array}{ccc} \Sigma_1 \cap \Sigma_2 & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_2 & \longrightarrow & \Sigma_1 \cup \Sigma_2 \ , \end{array}$$

where Σ_i is the signature defined by Sp_i, $i = 1, 2$. By the phrase '$\Sigma_1 \cup \Sigma_2$ is amalgamable' we mean that this diagram is amalgamable. However, there is no standard way of forming intersections of *specifications* in CASL. The model class of such an intersection would consist of all reducts of models of either Sp_1 or Sp_2 (such a structuring operation for specifications has been suggested in [3]);

this model class need not be specifiable in CASL. It suffices, however, to lift the above diagram of signatures to a diagram



of specifications where Sp_1 is conservative over Sp; these are precisely the requirements on Sp in the premises of the rule (union).

Example 1. Without amalgamability, instantiations of parametrized specifications may produce nasty surprises. Consider, e.g., the following specifications:

spec SP1 =
  sorts s, t
  op $f : s \to t$
  axiom $\forall x, y : s \bullet f(x) = f(y) \Rightarrow x = y$

spec SP2 =
  SP1
then
  sorts $s < t$
  op $g : s \to t$
  axiom $\forall x : s \bullet x = g(x)$

spec SP3[SP1]=
  sorts $s < t$
  axiom $\forall x : s \bullet x = f(x)$

The instantiation SP3[SP2] fails to be conservative over its argument SP2 (since the interpretations of f and g are forced to be equal), although the parametrized specification SP3[SP1] is conservative over its formal parameter SP1. Indeed, the signature diagram of the instantiation SP3[SP2] is the standard counterexample for amalgamation in CASL (cf. [21]).

A particular difficulty is attached to the **closed** construct, for which there is, as yet, no rule in the conservativity calculus. Dealing with this construct properly would require carrying around the local environment (cf. [7]), which appears, for the moment, to be an undue complication of the calculus. For practical cases, the equivalence rules of Figure 1 have proven to be sufficient.

Derived Rules One frequently used derived rule is

$$\textbf{(cext)}\ \frac{c(Sp_1),\ cons(Sp_1)(Sp_1\ \textbf{then}\ Sp_2)}{c(Sp_1\ \textbf{then}\ Sp_2)}$$

The derivation of this rule makes use of the rule (comp) of Figure 2. Moreover, using the existing conservativity rule for unions, one may derive

$$\textbf{(union2)}\ \frac{\begin{array}{c}cons(Sp_1)(Sp_1')\\ cons(Sp_2)(Sp_2')\\ \text{The signatures of } Sp_1' \text{ and } Sp_2' \text{ are disjoint}\end{array}}{cons(Sp_1\ \textbf{and}\ Sp_2)(Sp_1'\ \textbf{and}\ Sp_2')}$$

Finally, it is convenient to have the following rule, derived from rules (rp1) and (rp2) of Figure 2:

$$\textbf{(rpl)}\ \frac{\begin{array}{c}Sp_1 \simeq Sp_1'\\ Sp_2 \simeq Sp_2'\\ cons/def/implies(Sp_1)(Sp_2)\end{array}}{cons/def/implies(Sp_1')(Sp_2')}$$

Definitionality Rules A somewhat rudimentary definitionality calculus is shown in Figure 5. Its rules cover two syntactical patterns: operations or predicates that are defined by a term or a formula, respectively, at the time of their declaration (basic items of this type are called operation and predicate definitions, respectively, in [7]) are, of course, definitional. The same goes for operations that come with axioms which amount to a recursive definition. The precise meaning of the phrase 'Sp_2 recursively defines new operations and predicates over Sp_1' is as follows: all signature items newly declared in Sp_2 are either operations or predicates on existing sorts; for all these signature items, Sp_2 contains recursive definitions; Sp_2 does not contain any further axioms; and Sp_2 does not contain hiding or renaming.

Recursion is only possible over constructor-generated sorts. Unless these sorts are explicitly freely generated, it has to be guaranteed that the definitions are independent of the choice of representative (the same goes for the recursive definitions of equality or observers as required in rules (gtd1) and (gtd2) of Figure 3).

Proof obligations It is quite instructive to briefly summarize what kinds of proof obligations arise from applying the rules of the calculus to a CASL specification. Astonishingly, only a few rules depend on 'real' theorem proving. One such example is the rule (view): here one has to discharge the metapredicate $correct(\textbf{view}\ VN : Sp\ \textbf{to}\ Sp' = SM)$. Other 'real' proof obligations are generated by the well-definedness condition for recursive definitions on non-freely generated datatypes (rules (gtd1), (gtd2), and (def2)) and the congruence condition for equality on such datatypes (rules (gtd1) and (gtd2)).

The 'next hardest' type of proof obligation arises in the context of instantiations of parametrized specifications. Here, the premises require checking signature diagrams for amalgamability. As discussed above, this is algorithmically hard in theory, but decidable by a polynomial algorithm in the practically relevant cases.

$$\textbf{(tr)}\ \frac{\begin{array}{c}cons(Sp_1)(Sp_2)\\ injective(SM_2)\\ SM_1 \text{ is the restriction of } SM_2 \text{ to } Sp_1\end{array}}{cons(Sp_1\ \textbf{with}\ SM_1)(Sp_2\ \textbf{with}\ SM_2)}$$

$$\textbf{(hd)}\ \frac{}{cons(Sp\ \textbf{hide}\ SL)(Sp)} \qquad \textbf{(rv)}\ \frac{}{cons(Sp\ \textbf{reveal}\ SL)(Sp)}$$

$$\textbf{(free)}\ \frac{\begin{array}{c}horn(Sp)\\ Sp \text{ has a closed term for each sort}\end{array}}{c(\textbf{free}\ Sp)} \qquad \textbf{(local)}\ \frac{c(Sp_1\ \textbf{then}\ Sp_2)}{c(\textbf{local}\ Sp_1\ \textbf{within}\ Sp_2)}$$

$$\textbf{(view)}\ \frac{c(Sp'), correct(\textbf{view}\ VN : Sp\ \textbf{to}\ Sp' = SM)}{c(Sp)}$$

$$\textbf{(name1)}\ \frac{c(Sp)}{c(\textbf{spec}\ SN = Sp\ \textbf{end})}$$

$$\textbf{(name2)}\ \frac{c(\{Sp_1\ \textbf{and} \ldots \textbf{and}\ Sp_n\}\ \textbf{then}\ Sp)}{c(\textbf{spec}\ SN\ [Sp_1] \ldots [Sp_n] = Sp\ \textbf{end})}$$

$$\textbf{(name3)}\ \frac{c(\{Sp''_1\ \textbf{and} \ldots \textbf{and}\ Sp''_m\}\ \textbf{then}\ \{Sp_1\ \textbf{and} \ldots \textbf{and}\ Sp_n\}\ \textbf{then}\ Sp)}{c(\textbf{spec}\ SN\ [Sp_1] \ldots [Sp_n]\ \textbf{given}\ Sp''_1, \ldots, Sp''_m = Sp\ \textbf{end})}$$

$$\textbf{(inst2)}\ \frac{\begin{array}{c}SN[Sp_1] \ldots [Sp_n] = Sp\\ FA_i : Sp_i \to Sp'_i, i = 1, \ldots, n\\ cons(\{Sp_1\ \textbf{and} \ldots \textbf{and}\ Sp_n\})(\{Sp_1\ \textbf{and} \ldots \textbf{and}\ Sp_n\}\ \textbf{then}\ Sp)\\ \text{The instantiation diagram for } SN[FA_1] \ldots [FA_n] \text{ is amalgamable}\end{array}}{cons(Sp'_1\ \textbf{and} \ldots \textbf{and}\ Sp'_n)(SN[FA_1] \ldots [FA_n])}$$

$$\textbf{(inst3)}\ \frac{\begin{array}{c}SN[Sp_1] \ldots [Sp_n]\ \textbf{given}\ Sp''_1, \ldots, Sp''_n = Sp\\ FA_i : \{\{Sp''_1\ \textbf{and} \ldots \textbf{and}\ Sp''_n\}\ \textbf{then}\ Sp_i\} \to \{\{Sp''_1\ \textbf{and} \ldots \textbf{and}\ Sp''_n\}\ \textbf{then}\ Sp'_i\}, \quad i = 1, \ldots, n\\ cons(\{Sp''_1\ \textbf{and} \ldots \textbf{and}\ Sp''_n\}\ \textbf{then}\ \{Sp_1\ \textbf{and} \ldots \textbf{and}\ Sp_n\})\\ (\{Sp''_1\ \textbf{and} \ldots \textbf{and}\ Sp''_n\}\ \textbf{then}\ \{Sp_1\ \textbf{and} \ldots \textbf{and}\ Sp_n\}\ \textbf{then}\ Sp)\\ \text{The instantiation diagram for } SN[FA_1] \ldots [FA_n] \text{ is amalgamable}\end{array}}{cons(\{Sp''_1\ \textbf{and} \ldots \textbf{and}\ Sp''_n\}\ \textbf{then}\ \{Sp'_1\ \textbf{and}\ Sp'_n\})(SN[FA_1] \ldots [FA_n])}$$

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

Fig. 4. Rules for structuring constructs

$$\textbf{(def1)}\ \frac{BI \text{ is an operation or predicate definition}}{def(Sp)(Sp\ \textbf{then}\ BI)}$$

$$\textbf{(def2)}\ \frac{Sp_2 \text{ recursively defines new operations or predicates over } Sp_1}{def(Sp_1)(Sp_1\ \textbf{then}\ Sp_2)}$$

Fig. 5. Definitionality rules

Most of the rules, however, have only premises of a purely syntactical nature: $horn(Sp)$, $newSorts(s)(Sp)$, and $injective(Sp, SM)$ are typical examples. Thus, our calculus indeed reduces the amount of theorem proving required in consistency proofs.

3 Examples

In this section, we present selected consistency proofs for specifications taken from the CASL library of Basic Datatypes [19]. Besides providing a standard library, this library also illustrates how to write and structure specifications in CASL. All important features of CASL basic and structured specifications are used. Furthermore, the Basic Datatypes are the largest collection of CASL specifications currently available.

We start with a simple proof within the horn fragment, an example of a consistency proof that does not involve any theorem proving but which is based solely on our calculus and simple syntactical analysis. As a more advanced example, we prove the consistency of a specification for sets. This proof illustrates how to propagate consistency along the CASL structuring constructs, how to handle instantations, and how to deal with generated types. Finally, we present the consistency proof for a specification of characters — a quite large, if not overly difficult specification which can, using our calculus, be proven to be consistent nearly exclusively by syntactical analysis.

3.1 Consistency Proofs within the Horn fragment

Figure 6 shows typical specifications from the library ALGEBRA_I in the collection of Basic Datatypes [19]. Nearly all of this library's specifications are in positive Horn form. Their consistency proofs are similar to the one shown in Figure 7 for the specification GROUP: first, we unfold the specification definition using the rule (name1). The next step is to decode the consistency predicate into a proposition on conservativity (decode). Now we add the empty specification to the second argument, justified by rule (sep2) of the extension calculus and the general replace rule (rp2). This finally allows us to apply the rule (horn). As the empty specification has no signature, the second premise of this rule holds

trivially, and it remains to prove the metapredicate *horn*. Within the specification GROUP we obtain by skolemization that the axiom *inverse* is in positive Horn form. Since the specification MONOID is also in positive Horn form, we may now conclude that GROUP is consistent. Note that the proof of the metapredicate *horn* just involves syntax checks. Furthermore, note that the conservativity calculus does not flatten the original specification.

spec BINALG = **sort** *Elem*; **op** $\_\_ * \_\_ : Elem * Elem \to Elem$

spec SEMIGROUP = BINALG
then **op** $\_\_ * \_\_ : Elem * Elem \to Elem$, **assoc**

spec MONOID = SEMIGROUP
then **ops** $e : Elem$; $\_\_ * \_\_ : Elem * Elem \to Elem$, **unit** e

spec GROUP = MONOID
then **forall** $x : Elem \bullet \exists x' : Elem \bullet x' * x = e$ %(*inverse*)%

Fig. 6. Specifications within the Horn fragement of CASL

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{c}(\textbf{spec GROUP} = \dots)}{\mathrm{c}(\textbf{MONOID then forall } x : Elem \bullet \exists x' : Elem \bullet x' * x = e)}\ (\text{name1})}{\mathrm{cons}(\{\})(\textbf{MONOID then forall } x : Elem \bullet \exists x' : Elem \bullet x' * x = e)}\ (\text{decode})}{\mathrm{cons}(\{\})(\{\} \textbf{ then MONOID then forall } x : Elem \bullet \exists x' : Elem \bullet x' * x = e)}\ (\text{sep2, rp2})}{horn(\textbf{MONOID then forall } x : Elem \bullet \exists x' : Elem \bullet x' * x = e)}\ (\text{horn})$$

Fig. 7. Consistency proofs with **(horn)**

3.2 Consistency of the specification SET

The specification of sets in [19] is split up into two parts: GENERATESET (c.f. Figure 8) is concerned with sort generation, while SET (c.f. Figure 9) provides the typical operations and predicates on sets.

Note that the following consistency proof of SET does not 'import' *c*(GENERATESET). What we need instead is a statement concerning conservativity, namely that in GENERATESET the specifiation body conservatively extends the parameter. This illustrates again that – although we are primarily interested

in showing specifications to be consistent – our proofs are essentially about conservativity.

Again we start by unfolding the specification definition:

$$\frac{\text{c}(\textbf{spec}\ \textsc{Set}\ [\textbf{sort}\ \mathit{Elem}]\ \textbf{given}\ \textsc{Nat} = \dots)}{\begin{array}{c}\text{c}(\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\\ \textbf{then}\ \%\textbf{def}\dots\textbf{then}\ \%\textbf{def}\dots\textbf{then}\ \%\textbf{def}\dots\textbf{then}\ \%\textbf{implies}\dots\textbf{end})\end{array}}\ (\text{name3})$$

Now we show that the four extensions annotated with %**def** and %**implies**, resp., do not affect the consistency of the specification. To this end, our first step is to produce two goals by applying the derived rule (cext) to the last extension:

$$\frac{\begin{array}{c}\text{c}(\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\\ \textbf{then}\ \%\textbf{def}\dots\textbf{then}\ \%\textbf{def}\dots\textbf{then}\ \%\textbf{def}\dots\textbf{then}\ \%\textbf{implies}\dots\textbf{end})\end{array}}{\begin{array}{c}\text{c}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ ),\\ \text{cons}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ )\ (\ \textsc{Nat}\dots\textbf{end})\end{array}}\ (\text{cext})$$

The first goal states that the first part of the specification up to the extension operator is consistent; the second claims that the extension preserves models.

For the moment, we consider only the second goal and – guided by the annotation %**implies**– strengthen it using the rules (def) and (imp):

$$\frac{\dfrac{\text{cons}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ )\ (\ \textsc{Nat}\dots\textbf{end})}{\text{def}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ )\ (\ \textsc{Nat}\dots\textbf{end})}\ (\text{def})}{\text{implies}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ )\ (\ \textsc{Nat}\dots\textbf{end})}\ (\text{imp})$$

Following our methodology, we assume here that this last predicate is discharged by a suitable theorem prover, which has to show that $\_\_\cup\_\_$ and $\_\_\cap\_\_$ are associative, commutative, and idempotent.

Thus, for the consistency proof of Set it remains to discharge

$$\text{c}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ ).$$

To this end, we apply rule (cext) to the last extension of this subspecification of Set and strengthen the second of the resulting goals using (def):

$$\frac{\text{c}(\ \textsc{Nat}\dots(M-N)\cup(N-M)\ )}{\begin{array}{c}\text{c}(\ \textsc{Nat}\dots \mathit{when}\ x\epsilon M\ \mathit{else}\ (\sharp M)+1\ ),\\ \text{def}(\ \textsc{Nat}\dots \mathit{when}\ x\epsilon M\ \mathit{else}\ (\sharp M)+1\ )\ (\ \textsc{Nat}\dots(M-N)\cup(N-M)\ )\end{array}}\ (\text{cext, def})$$

The second goal is discharged by syntactical analysis: as the operator $\_\_\mathit{symmDiff}\_\_$ is introduced by an operation definition, we can apply rule (def1).

The remaining proof obligation is

$$\text{c(}\ \textsc{Nat}\ \ldots \mathit{when}\ x \epsilon M\ \mathit{else}\ (\sharp M) + \mathit{1}\ \text{)}$$

The technique for discharging this goal is the same as in the previous two steps:

$$\frac{\text{c(}\ \textsc{Nat}\ \ldots \mathit{when}\ x \epsilon M\ \mathit{else}\ (\sharp M) + \mathit{1}\ \text{)}}{\begin{array}{c}\text{c(}\ \textsc{Nat}\ \ldots = M + x\ \text{),}\\ \text{def(}\ \textsc{Nat}\ \ldots = M + x\ \text{) (}\ \ \textsc{Nat}\ \ldots \mathit{when}\ x \epsilon M\ \mathit{else}\ (\sharp M) + \mathit{1}\ \text{)}\end{array}}\ \text{(cext, def)}$$

This time, the resulting proof obligation concerning definitionality is more interesting: using (def2) to show that the operators $\_\_\cup\_\_$, $\_\_\cap\_\_$, $\_\_-\_\_$, $\_\_-\_\_$, and $\sharp\_\_$ are definitional over the previous specification involves — besides syntactical analysis — some theorem proving: e.g., given the axioms for $\_\_ \cup \_\_$

- $M \cup \{\} = M$
- $M \cup (N + x) = (M \cup N) + x$,

one has to show that

- $\{\} = N + x \Rightarrow M = (M \cup N) + x$
- $N + x = O + y \Rightarrow (M \cup N) + x = (M \cup O) + y$

Next, we have to establish

$$\text{c(}\ \textsc{Nat}\ \ldots = M + x\ \text{).}$$

Here we again apply (cext) to the last extension, strengthen the second resulting goal by (def), and discharge it by syntactical analysis using rule (def1).

Now the consistency problem for the original specification is reduced to

$$\text{c(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\text{).}$$

By (cext), this again splits up into a consistency and a conservativity problem:

$$\frac{\text{c(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\text{)}}{\begin{array}{c}\text{c(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\text{),}\\ \text{cons(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\text{)}\\ \text{(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\text{)}\end{array}}\ \text{(cext)}$$

Assuming Nat to be consistent, we obtain from the rules (cext) and (horn) the consistency of Nat **then sort** *Elem*.

Thus it remains to deal with the second goal:

$$\begin{array}{c}\text{cons(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\text{)}\\ \text{(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\text{)}\end{array}$$

Using the extension calculus, we obtain the equivalence

$$\begin{array}{c}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}] \simeq\\ \textsc{Nat}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\end{array}$$

which allows us to modify our goal by rule (rp2) into

$$\text{cons(}\textsc{Nat}\ \textbf{then sort}\ \mathit{Elem}\text{) (}\textsc{Nat}\ \textbf{then}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}]\text{)}$$

The combination of rule (then-and) of the extension calculus and the derived rule (rpl) transforms this into

$$\text{cons}(\textsc{Nat}\ \textbf{and sort}\ \mathit{Elem})\ (\textsc{Nat}\ \textbf{and}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}])$$

As the signatures of NAT and GENERATESET [**sort** *Elem*] are disjoint, we can apply the derived rule (union2) in order to obtain:

$$\frac{\text{cons}(\textsc{Nat}\ \textbf{and sort}\ \mathit{Elem})\ (\textsc{Nat}\ \textbf{and}\ \textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}])}{\text{cons}(\textsc{Nat})(\textsc{Nat}),\ \text{cons}(\textbf{sort}\ \mathit{Elem})(\textsc{GenerateSet}\ [\textbf{sort}\ \mathit{Elem}])}\ \text{(union2)}$$

cons(NAT)(NAT) holds if *def* (NAT)(NAT) is true (def), which can be justified by *implies*(NAT)(NAT) (imp), which holds thanks to (triv).

Thus our remaining problem is:

$$\text{cons}(\textbf{sort}\ \mathit{Elem})(\textsc{GenerateSet}[\textbf{sort}\ \mathit{Elem}]).$$

Rule (inst2) deals with the instantiation of GENERATESET:

$$\frac{\text{cons}(\textbf{sort}\ \mathit{Elem})(\textsc{GenerateSet}[\textbf{sort}\ \mathit{Elem}])}{\begin{array}{c}\text{amalg. inst. diagr. for } \textsc{GenerateSet}[\textbf{sort}\ \mathit{Elem}],\\ \text{cons}(\textbf{sort}\ \mathit{Elem})\\ (\textbf{sort}\ \mathit{Elem}\ \textbf{then generated type}\ \mathit{FinSet}[\mathit{Elem}] ::= \dots)\end{array}}\ \text{(inst2)}$$

Obviously the instantiation diagram is amalgamable. Furthermore, we have to make sure that the specification body of GENERATESET is conservative over its parameter. (From a methodological point of view, in the consistency proof for SET, one would import the following proof as an already established result on GENERATESET.) This is achieved by applying rule (gtd2):

$$\frac{\begin{array}{c}\text{cons}(\textbf{sort}\ \mathit{Elem})\\ (\textbf{sort}\ \mathit{Elem}\ \textbf{then generated type}\ \mathit{FinSet}[\mathit{Elem}] ::= \dots)\end{array}}{\begin{array}{c}\mathit{newSorts}(\mathit{FinSet}[\mathit{Elem}])(\textbf{sort}\ \mathit{Elem})\\ \mathit{Sp}_2 \text{ recursively defines an observer } \_\_\epsilon\_\_\\ \mathit{Sp}_3 \text{ defines equality on new sort by } \_\_\epsilon\_\_\end{array}}\ \text{(gtd2)}$$

where Sp_2 is
pred $\_\_\epsilon\_\_ : \mathit{Elem} * \mathit{FinSet}[\mathit{Elem}]$;
forall $x, y : \mathit{Elem}; M, N : \mathit{FinSet}[\mathit{Elem}]$

- $\mathit{not}\ x\epsilon\{\}$
- $x\epsilon(M + y) \Leftrightarrow x = y \vee x\epsilon M$

and Sp_3 denotes
forall $M, N : \mathit{FinSet}[\mathit{Elem}]$

- $M = N \Leftrightarrow (\forall x : \mathit{Elem} \bullet x\epsilon M \Leftrightarrow x\epsilon N)$

Obviously, $FinSet[Elem]$ is a new sort over **{sort** $Elem$**}**, and the predicate $\_\_\epsilon\_\_$ is a recursively defined observer. Concerning the new equality on $FinSet[Elem]$, the proof obligation
forall $M, N : FinSet[Elem]$
• $(\forall x : Elem \bullet x\epsilon M \Leftrightarrow x\epsilon N) \Rightarrow (\forall x, y : Elem \bullet x\epsilon(M + y) \Leftrightarrow x\epsilon(N + y))$
has to be discharged. As this is a consequence of the definition of $\_\_\epsilon\_\_$, the consistency proof for SET is finished.

spec GENERATESET [**sort** $Elem$] =
  generated type $FinSet[Elem] ::= \{\} \mid \_\_ + \_\_(FinSet[Elem]; Elem)$;
then %**def**
  pred $\_\_\epsilon\_\_ : Elem * FinSet[Elem]$;
  forall $x, y : Elem; M, N : FinSet[Elem]$
  • $\neg\ x\epsilon\{\}$
  • $x\epsilon(M + y) \Leftrightarrow x = y \vee x\epsilon M$
then
  forall $M, N : FinSet[Elem]$
  • $M = N \Leftrightarrow (\forall x : Elem \bullet x\epsilon M \Leftrightarrow x\epsilon N)$
end

Fig. 8. The specification GENERATESET

3.3 Consistency of CHAR

The specification CHAR of [19] (c.f. Figure 10) consists of about 1000 lines of CASL text, most of them operation or predicate definitions. This allows us to reduce the consistency problem for this specification by systematic use of the rules (cext), (def), and (def1) to the consistency of
NAT
then sort $Byte = \{n : Nat \bullet n \leq 255\}$
then free type $Char ::= chr(ord : Byte)$
Again assuming NAT to be consistent, we can establish this using the rules (cext), (sub), and (free): obviously, $0 \leq 255$, so that there is at least one element in the carrier of **sort** $Byte$. The premises of (free) are discharged by syntactical analysis: $Char$ is a new sort, and $chr(0)$ is a closed term of sort $Char$.

Thus, theorem proving is necessary in just one step of our consistency proof, the others require only 'pattern matching', i.e. finding suitable rules to be applied to the specification text, and syntactical analysis (to discharge the premises of these rules). Dealing with a flat specification with about 800 different axioms just by theorem proving would be nearly impossible. One might argue that an 'intelligent' theorem prover would sort out the same 'core specification' by some kind of syntactical analysis as well — but this just illustrates the necessity of

spec SET [**sort** *Elem*] **given** NAT = GENERATESET [**sort** *Elem*]
then %**def**
  preds $isNonEmpty\ (M : FinSet[Elem]) \Leftrightarrow \neg\ M = \{\}$
    $\_\_ \subseteq \_\_\ (M, N : FinSet[Elem]) \Leftrightarrow \forall x : Elem \bullet x \epsilon M \Rightarrow x \epsilon N$
  ops $\{\_\_\}\ (x : Elem) : FinSet[Elem] = \{\} + x$
    $\_\_ + \_\_\ (x : Elem; M : FinSet[Elem]) : FinSet[Elem] = M + x$
then %**def**
  ops $\_\_ \cup \_\_, \_\_ \cap \_\_, \_\_ - \_\_ : FinSet[Elem] * FinSet[Elem] \to FinSet[Elem];$
    $\_\_ - \_\_ : FinSet[Elem] * Elem \to FinSet[Elem];$
    $\sharp\_\_ : FinSet[Elem] \to Nat;$
  forall $x, y : Elem; M, N : FinSet[Elem]$
  - $M \cup \{\} = M$
  - $M \cup (N + x) = (M \cup N) + x$
  - $M \cap \{\} = \{\}$
  - $M \cap (N + x) = M \cap N\ when\ \neg x \epsilon M\ else\ (M \cap N) + x$
  - $M - \{\} = M$
  - $M - (N + x) = (M - N) - x$
  - $\{\} - x = \{\}$
  - $(M + x) - y = M - y\ when\ x = y\ else\ (M - y) + x$
  - $\sharp\{\} = 0$
  - $\sharp(M + x) = \sharp M\ when\ x \epsilon M\ else\ (\sharp M) + 1$

then %**def**
  op $\_\_symmDiff\_\_\ (M, N : FinSet[Elem]) : FinSet[Elem] = (M - N) \cup (N - M)$
then %**implies**
  ops $\_\_ \cup \_\_, \_\_ \cap \_\_ : FinSet[Elem] * FinSet[Elem] \to FinSet[Elem],$
    assoc, **comm**, **idem**;
end

Fig. 9. The specification SET

the strategy pursued here: economical proof organization by replacing parts of the necessary theorem proving by the application of 'meta rules'.

spec CHAR = NAT
then sort $Byte = \{n : Nat \bullet n \leq 255\}$
then free type $Char ::= chr(ord : Byte)$
then %**def**
  ops $'\backslash 000' : Char = chr(0); \ldots; '\backslash 255' : Char = chr(255);$
then %**def**
  ops $' ' : Char = '\backslash 032'; \ldots; 'ÿ' : Char = '\backslash 255';$
then %**def**
  preds $isLetter(c : Char) \Leftrightarrow ((ord('A') \leq ord(c) \wedge ord(c) \leq ord('Z')) \vee (ord('a') \leq ord(c) \wedge ord(c) \leq ord('z')));$
  $isDigit(c : Char) \Leftrightarrow ord('0') \leq ord(c) \wedge ord(c) \leq ord('9');$
  $isPrintable(c : Char) \Leftrightarrow ((ord(' ') \leq ord(c) \wedge ord(c) \leq ord('\sim')) \vee (ord(' ') \leq ord(c) \wedge ord(c) \leq ord('ÿ')))$
then %**def**
  ops $'\backslash o000' : Char = '\backslash 000'; \ldots; '\backslash o377' : Char = '\backslash 255';$
  $'\backslash x00' : Char = '\backslash 000'; \ldots; '\backslash xFF' : Char = '\backslash 255';$
  $NUL : Char = '\backslash 000'; \ldots$
then %**def**
  ops $NL : Char = LF; \ldots$
then %**def**
  ops $'\backslash n' : Char = NL; \ldots; '\backslash ?' : Char = '?';$
end

Fig. 10. The specification CHAR

4 Conclusions and future work

We have developed a calculus for proving conservativity of specification extensions in CASL, and we have used this calculus to establish the consistency of substantial portions of the CASL Basic Datatypes [19] (which have a good claim to being the largest CASL specification presently in existence). Several examples of such proofs have been presented and discussed. These examples have illustrated how the calculus facilitates the exploitation of the specification structure for the structuring of proofs. In fact, the proofs were 'automatically' directed to the few critical items that required proper theorem proving; by contrast, most of the proof obligations produced along the way were of an entirely syntactical nature.

There is no claim that the calculus as presented here is complete (since the underlying logic of CASL is undecidable, absolute completeness cannot be expected anyway); however, the case study that has been conducted on the library of Basic Datatypes, which makes use of all important features of CASL

basic and structured specifications, has shown that the calculus is able to deal with quite substantial specifications. (More precisely, five sublibraries containing more than 80 specifications of an overall length of roughly 2500 lines have been checked for consistency.) Summing up, we believe that our calculus is able to deal with consistency problems that arise in the context of software engineering projects.

Although no explicit mention was made of the concept of institution [12], the parts of the calculus that concern the CASL structuring mechanisms are, just as these mechanisms themselves [8], in fact institution independent. This makes the calculus easily adaptable with respect to, e.g., extensions of the underlying logic.

Future directions of research include the development of the second method of 'verifying' specifications mentioned in the introduction, namely, the testing of intended consequences, as well as the implementation of tool support for the conservativity calculus; this will possibly involve use of the development graph [2, 17]. It is expected that a forthcoming tool will allow semiautomatic consistency proofs, with the syntactical premises discharged automatically — via syntactic analysis or the more complex algorithms discussed in [15] — and the 'hard' ones output as formal proof obligations.

References

[1] Wolfgang Ahrendt, *A basis for model computation in free data types*, Proceedings of the CADE-17 Workshop on Model Computation, 2000.

[2] S. Autexier, D. Hutter, H. Mantel, and A. Schairer, *Towards an evolutionary formal software development using* CASL, Recent Trends in Algebraic Development Techniques, LNCS, vol. 1827, Springer, 1999, pp. 73–88.

[3] H. Baumeister, *Relations between abstract datatypes modeled as abstract datatypes*, Ph.D. thesis, Universität des Saarlandes, 1998.

[4] M. Bidoit, M. V. Cengarle, and R. Hennicker, *Proof systems for structured specifications and their refinements*, Algebraic Foundations of Systems Specification (E. Astesiano et al., eds.), Springer, 1999, pp. 385–433.

[5] M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski, *Permissive subsorted partial logic in* CASL, Algebraic Methodology and Software Technology, LNCS, vol. 1349, Springer, 1997, pp. 91–107.

[6] CoFI, *The Common Framework Initiative for algebraic specification and development, electronic archives*, notes and documents accessible from http://www.brics.dk/Projects/CoFI.

[7] CoFI Language Design Task Group, CASL – *The* CoFI *Algebraic Specification Language – Summary, version 1.0.1*, Documents/CASL/Summary, in [6], March 2001.

[8] CoFI Semantics Task Group, CASL – *The* CoFI *Algebraic Specification Language – Semantics*, Note S-9 (version 0.96), in [6], July 1999.

[9] R. Diaconescu, J. Goguen, and P. Stefaneas, *Logical support for modularisation*, Logical Environments, Cambridge, 1993, pp. 83–130.

[10] J. Farrés-Casals, *Proving correctness of constructor implementations*, Mathematical Foundations of Computer Science, LNCS, vol. 379, Springer, 1989, pp. 225–236.

[11] J.-Y. Girard, *Locus solum*, Math. Struct. Comput. Sci., To appear.
[12] J. Goguen and R. Burstall, *Institutions: Abstract model theory for specification and programming*, J. ACM **39** (1992), 95–146.
[13] M. J. C. Gordon and T. M. Melham, *Introduction to HOL: A theorem proving environment for higher order logics*, Cambridge, 1993.
[14] R. Hennicker and M. Wirsing, *Proof systems for structured algebraic specifications: An overview*, Fundamentals of Computation Theory, LNCS, vol. 1279, Springer, 1997, pp. 19–37.
[15] B. Klin, P. Hoffman, A. Tarlecki, L. Schröder, and T. Mossakowski, *Checking amalgamability conditions for* CASL *architectural specifications*, Mathematical Foundations of Computer Science, LNCS, Springer, 2001, to appear.
[16] T. F. Melham, *A package for inductive relation definitions in HOL*, International Workshop on the HOL Theorem Proving System and its Applications, IEEE Computer Society Press, 1992, pp. 350–357.
[17] T. Mossakowski, S. Autexier, and D. Hutter, *Extending development graphs with hiding*, Fundamental Aspects of Software Engineering, LNCS, vol. 2029, Springer, 2001, pp. 269–283.
[18] W. Reif, G. Schellhorn, and A. Thums, *Flaw detection in formal specifications*, International Joint Conference on Automated Reasoning, LNCS, vol. 2083, Springer, 2001, pp. 642–657.
[19] Markus Roggenbach, Till Mossakowski, and Lutz Schröder, *Basic datatypes in CASL*, Note L-12 in [6], current version 0.7 available at http://www.informatik.uni-bremen.de/cofi/CASL/lib/basic, March 2001.
[20] Markus Roggenbach and Lutz Schröder, *Towards trustworthy specifications II: Testing by proof*, work in progress.
[21] L. Schröder, T. Mossakowski, and A. Tarlecki, *Amalgamation in* CASL *via enriched signatures*, International Colloquium on Automata, Languages and Programming, LNCS, vol. 2076, Springer, 2001, pp. 993–1004.
[22] J. R. Shoenfield, *Mathematical logic*, Addison-Wesley, 1967.